1 Description of the programming project

In this third programming project, we will continue developing the shell that we started in the first two projects. We will add to the shell two main functionalities: the capability to execute pairs of commands that communicate among them, and the capability of directly communicate with other process executing instances of the shell. **All previous functionality should continue to work correctly**. In addition to all commands from the previous project, the shell should process the following commands:

1. `pipe prog1 arg11 ... arg1n | prog2 arg21 ... arg2m`: This command executes `prog1` with arguments `arg11 ... arg1n` and `prog2` with arguments `arg21 ... arg2n` and waits until the completion of both (similar to the `fg` command from the previous project). `prog1` reads from the same standard input as the shell (like the `fg` command), but its standard output is redirected to the standard input of `prog2`. The standard output of `prog2` is the same as the standard output of the shell. This works as invoking
   ```
   prog1 arg11 ... arg1n | prog2 arg21 ... arg2m
   ```
   in the system shell. See Section 2.1 for more details

2. `send message [pri]`: This command sends `message` to any process executing shell that invokes the `receive` command below. The process invoking `receive` may be the same that invoked `send` or a different one. The argument `message` is a string of characters not containing blank spaces. If its length is greater than 8192, the following error should be reported
   ```
   send: message should contain no more that 8192 characters
   ```
   This command should allow to queue up to 10 messages, that would be delivered to the invoker of `receive`. If 10 messages are already queued when the command is invoked, it just blocks until there is enough space in the queue. The optional argument `pri` is an integer strictly greater than 0 representing the priority associated with the message.

3. `receive`: Prints a messages sent by this and other process executing the shell. Messages with higher priorities are printed first. Among messages with the same priority the older messages are printed first. If there are no messages, it blocks until a message is received. See Section 2.2 for more details

4. `incr N`: it increments a pair of internal counters by `N` and prints their current value in the following format:
   ```
   @> incr 1000
   open shells: 3
   increment counter (local): 5000
   increment counter address (local): 0x55ef4d18e108
   increment counter (global): 7030
   increment counter address (global): 0x7fc8ee281004
   ```
   In particular, this means that there are currently 3 process running instances of the shell, the counter was incremented in 5000 units in this instance of the shell and in 7030 when considering all the running instances. 0x55ef4d18e108 and 0x7fc8ee281004 respectively are the addresses in memory of these two counters. The argument `N` must be a positive integer strictly greater than 0. If this argument is omitted or it is present but it is anything else than a positive integer strictly greater than 0, the shell should report the following error:
   ```
   usage: incr N where N is an integer strictly greater than 0
   ```
   Note that to implement this command there must be communication between all processes running instances of the shell. See Section 2.3 for more details.

5. **sendh message**: This command sends **message** to any process executing shell that invokes the **receiveh** command below. Commands **sendh** and **receiveh** are similar to commands **send** and **receive** above, but they do not accept priorities and must be implemented in a different way. As above, the process invoking **receiveh** may be the same that invoked **sendh** or a different one. The argument **message** is a string of characters not containing blank spaces. If its length is greater than 8192, the following error should be reported

   send: message should contain no more that 8192 characters

   This command should allow to queue up to 10 messages, that would be delivered to the invoker of **receiveh**. If 10 messages are already queued when the command is invoked, it just blocks until there is enough space in the queue.

6. **receiveh**: Prints a messages sent by this and other process invoking **sendh**. Messages are printed in the order that they were sent. If there are no messages, it blocks until a message is received. See Section 2.4 for more details.

Error handling The shell must not crash if something unexpected occurs or a command cannot be performed for any reason. The shell must not fail to execute a command without reporting a reason. Instead an error message should be produced.

2 Clues and details

2.1 The `pipe` command

Implementing the command **pipe** requires combining the ideas used in commands **redirect** and **fg** from the previous project, together with the system call `pipe()`. The following code creates a pipe

```
int pipefd[2];
ERROR_CHECK(pipe(pipefd));
```

where `ERROR_CHECK()` encapsulates the error checking code. As an example `ERROR_CHECK()` can be a preprocessor macro defined as follows:

```
#define ERROR_CHECK(x) do{ \
    if ((intptr_t)x == (intptr_t)-1) { \
        perror("myshell"); \
        return 0; \
    } \
}while(0)
```

This would print the standard error using function `perror()` and return from the current function. Each of the elements of `pipefd` is a file descriptor and can be used as a regular file. Therefore, it can be used with the system call `dup2()` to perform a redirection. See the details of this system call in the man pages and an example of its use in the class slides. Remember to close all the file descriptors that each process does not need. Each of the processes executing a command should close one of the file descriptors and the shell itself should close both (after calling fork). All notes regarding redirection explained in project 2 also apply here. Read carefully the description of project 2.

2.2 Commands `send` and `receive` with message queues

The basic implementation of the **send** and **receive** commands use a message queue. You must name your message queue **/myshell_*login*_mqueue** where ***logic*** is your login name in **odin**.

The name of your message queue is extremely important as **odin** is a shared machine and any collision with other message queues in the system will prevent your shell from working properly. The message queue should be created when invoking **./myshell --init** and removed when when invoking **./myshell --destroy**. Recall that you may assume that when **./myshell --init** or **./myshell --destroy** are invoked no process in the system is executing your shell. Hence, no synchronization is needed when implementing their behavior. When the shell is invoked without parameters or with the parameter **--echo** any number of processes can be executed the shell. Hence synchronization is needed. When using message queues, the operating system takes care of the synchronization, so this does not require special attention from the programmer. Note also that when invoking **./myshell --init** and **./myshell --destroy** the shell should not show the prompt nor execute any commands. It just initializes or destroys the structures it needs for future work and exits.

A message queue is created using the following system call invocation

```
int oflags  = O_CREAT | O_RDWR;
mode_t mode = S_IRWXU | S_IRWXG | S_IRWXO;
ERROR_CHECK(mq_open(MQUEUE_NAME, oflags, &attr));
```

and can be removed from the system with the following invocation

```
ERROR_CHECK(mq_unlink(MQUEUE_NAME);
```

You can check that the message queue has been correctly created or removed with the command, **ls /dev/mqueue/**, which lists all message queues existing currently in the system. Argument `attr` in the call to `mq_open()` is used to set the following queue parameters

```
mq_maxmsg  = 10
mq_msgsize = 8192
```

specifying the maximum number of queued messages and the maximum size of each one of them. **Programs using message queues must be compiled with `cc -lrt` to link against the real-time library *librt*.** That is, the flag **-lrt** needs to be added to the flags you are currently using to compile your code.

2.3 Command `incr`

To implement command **incr**, two shared counters are needed: a counter for the number of shells and a counter for the global number of increments. Therefore, these two counters should be held in a shared region of memory. The share region of memory should be created when invoking **myshell --init** and removed when invoking **myshell --destroy**. As in the previous section, you can assume that only one process at a time is executing your shell when invoking **myshell --init** or **myshell --destroy**. When executing **myshell** without parameters or with the parameter **--echo**, you must consider that many processes can be running instances of the shell and synchronization is needed. Synchronization can be achieved by declaring these counters as `atomic_int` and use `atomic_fetch_add()` to increment their value. The following code declares the type of structure that contains these two counters:

```
typedef struct {
    atomic_int num_shells;
    atomic_int num_increments;
} myshell_shared_data_t;
```

and the following code atomically adds `value` to the counter `num_increments`

```
myshell_shared_data_t *shared_data;
... // initialization of the structure
atomic_fetch_add(&shared_data->num_increments, value);
```

The following code creates a shared region of memory and assigns it to the variable `shared_data`.

```
int oflags  = O_CREAT | O_RDWR;
mode_t mode = S_IRWXU | S_IRWXG | S_IRWXO;
int prot    = PROT_READ | PROT_WRITE;
size_t size = sizeof(mysell_shared_data_t);
shm_fd = shm_open(SHM_NAME, oflags , mode);
ERROR_CHECK(shm_fd);
ERROR_CHECK(ftruncate(shm_fd, sizeof(mysell_shared_data_t)));
shared_data = mmap(0, size, prot, MAP_SHARED, shm_fd, 0);
ERROR_CHECK(shared_data);
```

After creation other processes can open the shared region with the following code:

```
int oflags  = O_RDWR;
int prot    = PROT_READ | PROT_WRITE;
size_t size = sizeof(mysell_shared_data_t);
shm_fd = shm_open(SHM_NAME, oflags, 0);
ERROR_CHECK(shm_fd);
shared_data = mmap(0, size, prot, MAP_SHARED, shm_fd, 0);
ERROR_CHECK(shared_data);
```

The shared region can be removed with the following code:

```
ERROR_CHECK(shm_unlink(SHM_NAME));
```

Addresses of counters can be printed with a code similar to the following:

```
printf("increment␣counter␣address␣(local):␣%p\n", &increments);
```

2.4 Commands `sendh` and `receiveh` with shared memory

The final piece of the project is an implementation of commands **`sendh`** and **`receiveh`** using the producer-consumer code implemented in shared memory. This version of commands **`sendh`** and **`receiveh`** should be used when the shell is initialized with the optional argument **`pc`**, that is by invoking **`myshell --init pc`**. When the optional argument **`pc`** is used in the initialization no message queues can be created. **The use or creation of message queues in the implementation of this version will yield a 0 score for these commands.** Be sure that no message queues created by your code are available when running the shell initialized with this argument. Commands **`send`** and **`receive`** should report and error when invoked if the shell was initialized with the argument **`pc`**.

We will divide the implementation of this functionality in four stages:

1. Single process: the sender and the receiver are the same process. No shared memory of synchronization is necessary.

2. Several synchronized processes: there may an arbitrary number of senders and receivers. It may be assumed that senders and receivers will not be invoking their respective commands at the same time. Therefore, shared memory is necessary, but not synchronization.

3. Two unsynchronized processes: there is a unique sender and a unique receiver that can be the same process or two different processes. The sender and the receiver may be invoking their respective commands at the same time. Therefore, shared memory and synchronization are necessary.

4. Several unsynchronized processes: there may an arbitrary number of senders and receivers that can be invoking their respective commands at the same time. Shared memory and synchronization are necessary.

Each stage will grant partial credit and it is possible to reach the full grade by correctly implementing stage 2 (see Section 5 for more details).

Stage 1: single process. To succeed in this stage, it is necessary to implement a circular or ring buffer with 10 elements of size 8192. The buffer is an array that can be declared as a local variable or stored in shared memory (the later is necessary for latter stages). The following is an example of structure that can be used to implement the ring buffer.

```
typedef struct {
    int count;
    int in;
    int out;
    char buffer[MSG_QUEUE_MAX][MSG_SIZE_MAX];
} pc_buffer_t;
```

where `count` represents the number of messages in the buffer, `in` is the position in the buffer where the next message will be inserted and `out` is the position in the buffer of the oldest message not printed yet. The invocation of command **`sendh message`** should add a message to the buffer and the invocation of **`receiveh`** should remove a message from the buffer. To copy the messages into/from the buffer you can use the function `memcpy()`.

Stage 2: two synchronized processes. To succeed in the second stage, the ring buffer structure simply needs to be put in a shared region of memory. This can be achieved by adding the ring buffer as a field to `myshell_shared_data_t` data type introduced in Section 2.3.

Stages 3 and 4: unsynchronized processes. To succeed in these two stages, synchronization needs to be added. Any of the methods saw in class for multiple producer-consumers will work. The method for single producer-consumer will work for stage 3. The clues provided here follow the monitor implementation simulated by careful use of pthreads `mutex` lock and `conditions`. This implementation works for both phases, but partial credit for succeeding only in stage 3 will be granted. For synchronization, a mutex and two conditions will be needed.

```
pthread_mutex_t mutex;
pthread_cond_t full;
pthread_cond_t empty;
```

These three variables can be added as fields to `pc_buffer_t`. The `mutex` and `conditions` must be initialized during the invocation of **`myshell --init pc`**. Default attributes will not suffice as they only allow them to be used among the threads of the same process. To allow `mutex` and `conditions` to be used among multiple processes, they must be initialized with `PTHREAD_PROCESS_SHARED`. The following code can be use to set the initialization.

```
pc_buffer_t pc_buffer;
// initialization of pc_buffer...

pthread_mutexattr_t mtx_at;
ERROR_CHECK0(pthread_mutexattr_init(&mtx_at));
ERROR_CHECK0(pthread_mutexattr_setpshared(&mtx_at, PTHREAD_PROCESS_SHARED));
ERROR_CHECK0(pthread_mutex_init(&pc_buffer->mutex, &mtx_at));

pthread_condattr_t cnd_at;
ERROR_CHECK0(pthread_condattr_init(&cnd_at));
ERROR_CHECK0(pthread_condattr_setpshared(&cnd_at, PTHREAD_PROCESS_SHARED));
ERROR_CHECK0(pthread_cond_init(&pc_buffer->full, &cnd_at));
ERROR_CHECK0(pthread_cond_init(&pc_buffer->empty, &cnd_at));
```

Note that `ERROR_CHECK0()` is different from the macro `ERROR_CHECK()` defined above as the error conditions of these methods are different. Check the corresponding manpages to correctly capture the errors these functions may return.

Methods for adding and removing messages from the buffer need to be protected by the `mutex` lock. For instance, the following is a possible template for inserting messages into the buffer:

```
int pc_insert(pc_buffer_t *pc_buffer, char *msg, int msg_size) {
    ERROR_CHECK0(pthread_mutex_lock(&pc_buffer->mutex));
    // code for inserting the message in the buffer here
    ERROR_CHECK0(pthread_mutex_unlock(&pc_buffer->mutex));
    return 0;
}
```

Remember that `pthread_cond_wait()` does not guarantee that only one thread is awaken. As a result, the condition must be rechecked after the thread is awaken. For instance, the following code can be used to block when the buffer is full.

```
while(pc_buffer->count >= MSG_QUEUE_MAX){
    ERROR_CHECK0(pthread_cond_wait(&pc_buffer->full, &pc_buffer->mutex));
}
```

For using pthreads mutex locks and conditions, it is necessary to add the flag `-lpthread` to the arguments of the compiler.

3 Submission

The submission must consist of a single file named exactly **`p3.tar.gz`** and should be submitted through the **`canvas`** web platform. Check in canvas the deadline for the submission. The file **`p3.tar.gz`** can be created with the command

```
tar -czvf p3.tar.gz *
```

After executing this command a file **`p3.tar.gz`** will be created containing all the files and directories in the current directory. The content of the file can be extracted with the command

```
tar -xzvf p3.tar.gz
```

All the content of **`p3.tar.gz`** will be extracted in the current directory. The file **`p3.tar.gz`** must contain a directory called **`src/`** and a file called **`Makefile`**. It may contain in addition a directory called **`test/`**. Inside the directory **`src/`** should be all the source code of your shell. Executing the command **`make`** should:

- compile your code,
- create a directory called **bin/**, and
- place a command called **myshell** inside it.

After executing the command **make**, executing the command **./bin/myshell** should execute your shell.

An example file **p3.tar.gz** can be download from the canvas platform. It contains the needed structure, **Makefiles** and a dummy **myshell.c** file. Inside the **test/** directory there is a bash script **compile_test.sh** that can be used to automatically test that these steps work correctly. To check that your **p3.tar.gz** file can be extracted and compiled correctly, copy it together with the script **compile_test.sh** into an empty directory; then execute the script. No error should be reported. **Projects that fail this test will get a 0 score.**

4 Automatic testing

The sample file **p3.tar.gz** posted on canvas also includes a series of scripts for automatic testing. These scripts are there to help you checking your progress. There is also a script named **all_test.sh** that runs all the other scripts and provides a summary of the results. You can run all the test by typing **test/all_test.sh** in the system shell. In case of an error on any of the tests, you can run that test individually. For instance, if test **cdir_test.sh**, then you can invoke **test/p1/cdir_test.sh**. After invoking an individual test, you can check the files **output.txt**, **error.txt**, **expected_output.txt**, **expected_error.txt**, **output_ls.txt** and **expected_ls.txt** in the directory **test/** to see the difference between your **shell** results and the expected ones. These files are not available after running **all_test.sh**. These tests will also be used for evaluating your project (see Section 5 below). If you believe that your shell is producing the correct result according to the above specification and yet the automatic test reports an error, then contact the instructor before submitting the project.

5 Evaluation

Be sure to test your code in the school **odin server** before submitting the exercise: `https://www.unomaha.edu/college-of-information-science-and-technology/about/odin.php`

Your program will be evaluated by testing a variety of commands. To receive 100 percent, your program must process the entire set of the given commands correctly. If **any of the following errors** occur, the exercise **will get a 0 score**:

- The program does not compile or execute on the odin server following the instructions in Section 3. In particular, if the **compile_test.sh** does not run correctly as specified in that section.
- The program executes, but it produces a runtime error (like segmentation-fault, bus error...)

The project will be evaluated through (i) automatic testing and (ii) manual/visual code inspection. **Note:** If the provided automatic test for a command fails, then the code implementing that command will not be inspected. The following table can be used to compute the maximum score expected for the functionality of project 3:

test	maximum score
`p3/pipe_test.sh`	20%
`p3/msg_queue_test.sh`	20%
`p3/incr_test.sh`	20%
`p3/pc_test_stage_01.sh`	10%
`p3/pc_test_stage_02.sh`	20%
`p3/pc_test_stage_03.sh`	10%
`p3/pc_test_stage_04.sh`	20%
total	120%

Extra tests will be executed to determine your final grade. You are encouraged to exhaustive test your code and write your own automatic tests. Note that the possible total score exceeds 100%. The extra score can be used to compensate for credit lost in other parts of the course (including homework and exams). That is, it is possible to earn more than 40% of the total score of the course with programming projects and this will be added to the scores obtained in the exams (40%) and the homework assignments (20%).

Continuous evaluation. This is the third iteration of the shell. As mentioned in the beginning of the document, **all previous functionality should continue to work correctly**. For the evaluation, this means that if any functionality that worked correctly in any of the first two iterations (the submission of the first or second project) does not work in this second submission, then a 20% of the grade obtained from it in these projects would be substracted from this third project. In no case the grade of the first project will be reduced.

On the other hand, you may receive up to 50% of the grade corresponding to any of the following tests of the first project that did not work in the previous submission.

`create_test.sh`, `delete_test.sh`, `hist_test.sh` and `list_test.sh`

This means that you can get up to 40% of the total grade for the first project in this submission. The test `cdir_test.sh` is not graded in this submission and it was provided as part of the a shell that you can use as base for the second project. Similarly, you may receive up to 75% of the grade corresponding to any of the following tests of the second project that did not work in the previous submission.

`redirect_test.sh`, `fork_test.sh`, `exec_test.sh`, `fg_test.sh` and `bg_test.sh`

The script `test/all_test.sh` has been updated to test for the commands of all the tree projects. The script `test/p3/all_test.sh` can be used to run the commands of the third project alone. Be sure to run `compile_test.sh` and `test/all_test.sh` before submission!