Operating Systems Architecture

Haiku project

This is the project for Operating Systems Architecture course of:

- Abbas Aliyev
- Aziz Salimli
- Habil Abdulkhaligov
- Narmin Pashayeva
- Nijat Hamidov

Introduction

The goal of this project is to demonstrate our skills in using operating systems' tools and interfaces by building an application that utilises them. Application consists of two separate parts: client and a server. Server will perform an action: display a haiku the user. Server is activated by request from a client: signal. We have 3 versions that show different OS tools.

Execution

To execute our haiku project you can use `make` system. To run each version we have created convenient rules. For version 1 and version 3 default is two executables build.

```
make version_1
```

```
make version_2
```

```
make version_3
```

Version 1

Version 1 of our haiku application will work with signals. There should be two processes: client and a server. Client sends one of two controlling signals to a server: SIGINT of SIGKILL, which correspond to the japanese or the western haiku, and SIGCHLD that signals stop of execution. Client version 1 will just send 100 random signals: SIGINT or SIGQUIT and SIGCHLD after that to signal termination. Server, depending on a signal, will print corresponding haiku. To communicate, client needs to know process id of a server, for this problem, our team has proposed two solutions, which resulted in two execution modes.

1. As single executable
2. As two executables

Single executable

When running as one executable, we are spawning daughter process using `fork()`, which will be our server. Parent process, which is client, gets child's process id, and will send signals to it. This way we start as one executable, but run 2 processes. This is

one way to pass server's process id to client.

Two executables

If we are running server and a client separately, we should have a way to run them first. For this to happen, there should exist main function in server.c and client.c. But if we just create a `int main(){...}` in server.c and/or client.c, it would conflict with main in main.c and won't let us run project as single executable. To resolve this issue, we came up with creative solution: using macros and defining them as argument to gcc. In both server.c and client.c we actually have main function, but it stays hidden until needed. In server main hides under alias s_main and in client as c_main. That's how neither client, nor server interfere with main.c's main function. But, we still have a way to run client and server separately, we just need to replace s_main and c_main with main and compile sources separately. This lines of code make this possible:

```
// server.c
#ifdef STANDALONE
#define s_main main
#endif
```

```
// client.c
#ifdef STANDALONE
#define c_main main
#endif
```

```
# makefile
server: server.h server.c queue.h queue.c haiku.h haiku.c
    $(CC) $(CFLAGS) -DSTANDALONE server.c queue.c haiku.c -o server $(LDFLAGS)

client: client.c client.c queue.h queue.c haiku.h haiku.c
    $(CC) $(CFLAGS) -DSTANDALONE client.c queue.c haiku.c -o client $(LDFLAGS)
```

We compile server and client separately and define macro STANDALONE with flag -D to gcc, which enables replacement mechanism of main functions.

haiku.h

To represent haiku in our program, we decided to create a special structure for it.

```
typedef struct haiku_{
    char author[64];
    char lines[3][64];
} haiku;
```

Haiku basically is a short, 3-line poem, written by some author. That's what `struct haiku_` represents: author string and 3 strings for each line. We decided to go with strings of moderate size, as static arrays. 63 chars + \0 char is enough for any haiku and author we are working with.
At the same time, we have two distinct groups of haiku: japanese and western. It is convenient to name these groups and represent them in our program. For that we have created category enumeration and book structure

```
typedef struct book_{
    int size;
    haiku *poems;
}book;

typedef enum category_{
    japanese,
    western
} category;
```

`struct book_` is a simple dynamic array. It stores number of haiku in it and pointer to the start of array of haiku. Book can be read from a file using `read_category()` or `read_book()` function:

```
struct book_ read_category(enum category_ c);
struct book_ read_book(const char *filename);
```

For each category, there is a corresponding file, that stores haiku of that category. Filenames are hardcoded into the list, that `read_category()` uses and passes to `read_book()`.

Version 2

Version 2 required creating methods of writing and reading haiku to the message queue and using them in reader and writer threads. Both reader and writer will be parts of server in the future. Here writer thread will send 3 haiku of each type to message queue and reader thread will read 3 haiku of each type. We have grouped all queue interactions to queue.h header file.

queue.h

In this header we defined all means and method of interaction with message queue. Message queue accepts data in special format: structure, that has the first element of type `long` and the payload. In our case payload is single haiku.

```
struct haiku_msg{
    long type;
    struct haiku_ package;
};
```

Then 3 controlling and 2 interacting function were defined:

```
// control
int create_queue();
int access_queue();
int remove_queue(int id);
// interactions
int write_haiku(category c, haiku *h);
int read_haiku(category c, haiku *h);
```

Create and access function are nothing special, they just use `ftok()` on working directory and 2 as project id to get unique queue id. Interesting part is `read_haiku()` function. It uses `msgrcv()` with `IPC_NOWAIT` flag. This make `msgrcv()` return immediately, whether

queue is full or empty. But if the queue is empty, it returns `-1` and `errno` is set to `ENOMSG`. That's how we detect if queue is empty and needs refilling.

Version 3

Version 3 is amalgamation of version 1 and version 2. Here we have simple client: client sending signals to server. And compound server: server runs 3 treads: reader, writer and main thread. Main thread gets signal from the client and passes it to the reader thread. Reader thread tries reading a haiku from a queue and display it to the user. If it fails, it sends a signal to writer thread, to refill the queue, then it sends a signal to reader thread again, for it to read from refilled queue. Server also react to `SIGCHLD`. This signal signals server to terminate. Same signal is passed to reader and writer thread, also terminating them.
Same mechanism of different execution modes from version 1 is available in version 3.

Tests

We are running test for all basic function: testing for reads, testing for message queue, testing for threads. Also, we are testing `read_category()` function for ensuring that we read book correctly.