Task description

Introducing ProgExchange (PEX), a cutting-edge platform for computer programmers to buy and sell high-demand components in a virtual marketplace. The need for in-person trading has been replaced by a state-of-the-art digital marketplace, allowing for greater accessibility and faster transactions, while providing a safe and convenient way for seasoned programmers to connect and make transactions.

As part of this assignment, you will be tasked with developing two key components of ProgExchange: the exchange itself, which will handle all incoming orders, and an auto-trading program that executes trades based on predefined conditions. With your expertise in systems programming, you will play a crucial role in bringing this innovative platform to life and helping to drive the future of computer component trading.

You are encouraged to ask questions on Ed[1]. Make sure your question post is of "**Question**" post type and is under "**Assignment**" category → "**A3**" subcategory. As with any assignment, make sure that your work is your own[2], and that you do not share your code or solutions with other students.

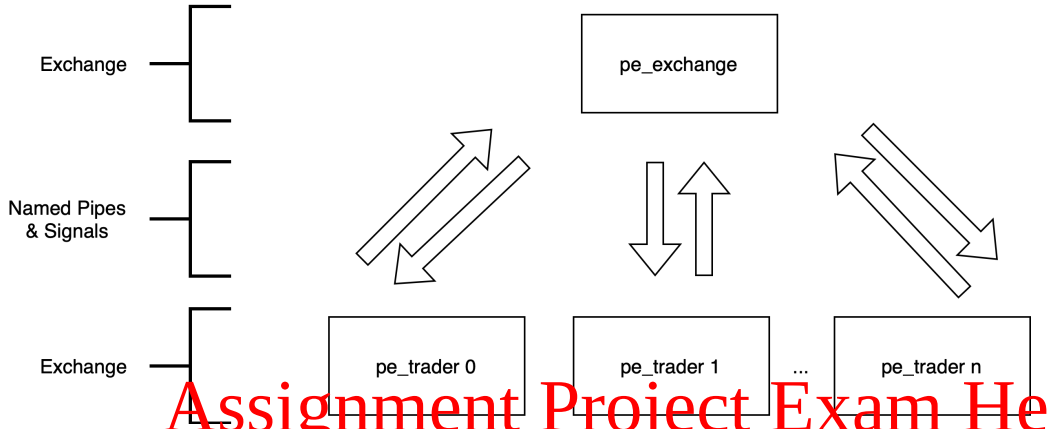
To complete this assignment, you should be familiar with:

- Dynamic memory allocation: malloc(), realloc(), free(), etc
- open(), read(), write() system calls
- Processes: fork(), exec(), waitpid(), etc
- Signals: sigaction(), kill(), etc
- Named Pipes (FIFOs): mkfifo()

[1] https://edstem.org/au/courses/10466/discussion/
[2] Not GPT-3/4's, ChatGPT's or copilot's, etc.

Some implementation details are purposefully left ambiguous; you have the freedom to decide on the specifics yourself. Additionally this description does not define all possible behaviour that can be exhibited by the system; some error cases are not documented. You are expected to gracefully report and handle these errors yourself.

High Level Overview



Exchange

The purpose of the ProgExchange is to allow trading to happen in an efficient and orderly manner. It receives orders from traders and matches them, allowing the buying and selling of computer components.

The exchange also tracks the cash balance of each trader (which starts at $0 for each trading session).

For providing such trading avenue, ProgExchange collects a 1% transaction fee on all successful orders.

IPC

The exchange communicates with trader processes using a combination of named pipes (FIFOs) and signals. All messages are followed by the delimiter `;` and signal `SIGUSR1`.

All messages sent through FIFOs are highlighted in this document:

```
EXAMPLE;
```

Traders

Traders carry out their buying and selling activities by placing orders on the exchange.

Commands

The following commands may be sent from traders to the exchange:

```
BUY <ORDER_ID> <PRODUCT> <QTY> <PRICE>;
SELL <ORDER_ID> <PRODUCT> <QTY> <PRICE>;
AMEND <ORDER_ID> <QTY> <PRICE>;
CANCEL <ORDER_ID>;
```

- **BUY**: An order to buy a product at or below the specified price, up to the specified quantity.
- **SELL**: An order to sell a product at the specified price, up to the specified quantity.
- **AMEND**: Update the quantity or price of an existing order, that has yet to be fully filled.
- **CANCEL**: Cancel an existing order, that has yet to be fully filled.

Data Types and Ranges

- **ORDER_ID**: integer, 0 - 999999 (incremental)
  Order ID is unique per Trader (i.e. Trader 0 and 1 can both have their own Order ID 0). Order IDs are not reused (with the exception of Invalid orders, which can be fixed and re-sent with the same ID, given the next ID is not yet used).

- **PRODUCT**: string, alphanumeric, case sensitive, up to 16 characters
- **QTY**, **PRICE**: integer, 1 - 999999

Products

Products traded on the exchange are provided through a text file of the following structure:

```
n_items
Product 1
Product 2
...
Product N
```

For example:

```
2
GPU
Motherboard
```

Basic Example

- Trader 0 places a SELL order for 15 CPUs at $300 each

```
SELL 0 CPU 15 300;
```

- Trader 1 places a BUY order for 10 CPUs at $300 each

```
BUY 0 CPU 10 300;
```

- ProgExchange matches these orders, Trader 1 buys 10 CPUs from Trader 0 for $3,000 and pays $30 transaction fee.
- Trader 1's order is now fulfilled. Trader 0 has 5 CPUs remaining on the market for sale.

Implementation details

Write programs in C that implement ProgExchange as shown in the examples.

You are guaranteed not to have NULL returned from `malloc()` or `realloc()` in this context.

Your submission must be contained in the following files and produce no errors when built and run on Ed C compiler.

- `pe_common.h`: Common constants and structures
- `pe_exchange.c`, `pe_exchange.h`: Part 1 (Exchange, compiles to `pe_exchange`)
- `pe_trader.c`, `pe_trader.h`: Part 2 (Trader, compiles to `pe_trader`)
- Test cases in `tests` directory
- `README.md`: Code Description

Read / write with FIFOs and/or write to stdout as instructed.

Your program output must match the exact output format shown in the examples and on Ed. You are encouraged to submit your assignment while you are working on it, so you can obtain feedback.

You may modify any of the existing scaffold files, or make your own.

No external code outside the standard C library functions should be required.

In order to obtain full marks, your program must free all of the dynamic memory it allocates.

Milestone: Auto-Trader

To complete the milestone, you need to implement an auto-trader in `pe_trader.c` and `pe_trader.h`, which will be tested against a reference exchange server implementation.

The logic of the auto-trader is very simple: as soon as a SELL order is available in the exchange, it will place an opposite BUY order to attempt to buy the item.

As an example, as soon as the auto-trader receives the following message (and signal) from the exchange:

```
MARKET SELL CPU 2 300;
```

It would place the following order:

```
BUY <Next Order ID> CPU 2 300;
```

The auto-trader should gracefully disconnect and shut down if there is a BUY order with quantity greater than or equal to 1000.

For the milestone, you may assume that for the purposes of this auto-trader, there are no other competing traders placing BUY orders.

However, signals are inherently unreliable, that is, multiple signals (perhaps from multiple auto-traders) may overwrite one another and cause a race condition. In the final version, you should design your auto-trader in such way that it is fault-tolerant, while conforming to the Exchange messaging protocol (first write to the pipe, then signal `SIGUSR1`), and does not place unreasonable load on the exchange.

Exchange: Start Up

The exchange accepts command line arguments as follows:

```
./pe_exchange [product file] [trader 0] [trader 1] ... [trader n]
```

The following example uses a product file named products.txt and trader binaries trader_a and trader_b:

```
./pe_exchange products.txt ./trader_a ./trader_b
```

Upon start up, the exchange should read the product file to find out about what it will trade. It should then create two named pipes for each trader:

```
/tmp/pe_exchange_<Trader ID>
/tmp/pe_trader_<Trader ID>
```

The pe_exchange_* named pipes are for the exchange write to each trader and pe_trader_* named pipes are for each trader to write to the exchange.

After both pipes are created for a trader, the exchange shall launch that trader binary as a new child process, assigning it a trader ID starting from 0, in the Command Line Argument order. The trader ID shall be passed to the trader binary as a command line argument.

For the example above, the trader binaries should be launched like so:

```
./trader_a 0
./trader_b 1
```


Upon launching each binary, the exchange and trader should attempt to connect to both named pipes.

After all binaries are launched and pipes are connected, the exchange should send each trader (lowest trader IDs first) the following message using the pe_exchange_* named pipes; afterwards, notify each trader using the `SIGUSR1` signal.

```
MARKET OPEN;
```

Exchange: Placing Orders

Traders may place orders with the exchange by sending the appropriate commands through their pe_trader_<Trader ID> named pipe. Once the whole command is written to the pipe, it shall notify the exchange with the `SIGUSR1` signal.

Once the exchange receives the `SIGUSR1` signal from a trader, it would read the command from the pe_trader_<Trader ID> named pipe and process the order appropriately. Depending on whether the order was accepted (for new buy / sell orders), amended or cancelled, or if the command is invalid, it would write one of the following messages to the pe_exchange_<Trader ID> named pipe and notify the trader using the `SIGUSR1` signal.

```
ACCEPTED <ORDER_ID>;
AMENDED <ORDER_ID>;
CANCELLED <ORDER_ID>;
INVALID;
```

The exchange should also message all other traders (lowest trader IDs first) using the pe_exchange_* named pipes, and notify them using the `SIGUSR1` signal, with the following message:

```
MARKET <ORDER TYPE> <PRODUCT> <QTY> <PRICE>;
```

N.B.: In case of a cancelled order, QTY = 0 and PRICE = 0.

Sample order flow

1. Trader 0 writes to pe_trader_0:

   ```
   SELL 20 CPU 2 300;
   ```


2. Trader 0 sends `SIGUSR1` signal to the Exchange
3. Exchange reads pe_trader_0 and adds the order to the order book

4. Exchange writes to pe_exchange_0:

   ```
   ACCEPTED 20;
   ```


5. Exchange sends `SIGUSR1` signal to Trader 0
6. Exchange writes to all other pe_exchange_* pipes:

   ```
   MARKET SELL CPU 2 300;
   ```

7. Exchange sends `SIGUSR1` signal to all other Traders.
8. All traders can read their own pe_exchange_* and places further orders if desired

Exchange: Matching Orders

The exchange should attempt to match orders once there is at least one BUY and one SELL order for any particular product. Orders match if the price of the BUY order is larger than or equal to the price of the SELL order. The matching price is the price of the older order. Of the two matched traders, the exchange charges 1% transaction fee to the trader who placed the order last, rounded to the nearest dollar (eg: $4.50 -> $5.00).

When there are multiple orders in the exchange, order matching should follow price-time priority:

- Match the highest-priced BUY order against lowest-priced SELL order

- If there are multiple orders at the same price level, match the earliest order first

As the order matches, the exchange shall write the following message to the appropriate pe_exchange_* pipes belonging to the matched traders, then send the `SIGUSR1` signal:

```
FILL <ORDER_ID> <QTY>;
```

It is possible for a single order with sufficient quantity to match against multiple existing orders in the market. Similarly, an order could be partially filled and remain in the market if there isn't sufficient quantity available. Orders are removed from the market once their quantity reaches zero.

Example Scenarios

Example	Orderbook Before (type, qty, price)	New Order	Orderbook After (type, qty, price)	Explanation
0	SELL 2 $500 BUY 2 $450	BUY 2 $500	BUY 2 $450	The new buy order matched against the SELL order. Both orders are fully filled.
1	SELL 2 $501 SELL 2 $500	BUY 5 $505	BUY 1 $505	The new BUY order matched against both SELL orders. The SELL orders are fully filled. The BUY order is partially filled (qty 4) and remains on the market (qty 1).

Exchange: Reporting

In addition to reading and writing to the named pipes, the exchange also needs to print a range of messages to standard out (stdout), as per the examples later in this document. Please follow the examples for the outputs required.

Specifically, the order book and trader positions need to be printed after each successful order, for example:

```
[PEX]   --ORDERBOOK--
[PEX]   Product: GPU; Buy levels: 3; Sell levels: 1
[PEX]           SELL 99 @ $511 (1 order)
[PEX]           BUY 30 @ $502 (1 order)
[PEX]           BUY 60 @ $501 (2 orders)
[PEX]           BUY 30 @ $500 (1 order)
<repeat for further products>
[PEX]   --POSITIONS--
[PEX]   Trader 0: GPU 0 ($0), Router 0 ($0)
<repeat for further traders>
```

Here, all orders in the market are sorted from the highest to lowest price. Each unique price is called a level, and there may be multiple orders in the same level. Positions refer to the quantity of products owned (or owed) by each trader, which may be positive or negative after each order.

Note: Tabs (`\t`) are used for indentation.

Exchange: Teardown

As soon as a trader disconnects (closing their end of the pipes or process exits), your exchange should print out the following message:

```
[PEX] Trader <Trader ID> disconnected
```

It shall reject any pending or further orders from the trader but keep existing orders in the orderbook (if any).

After all traders disconnect, the exchange should print out the following message:

```
[PEX] Trading completed
[PEX] Exchange fees collected: $<total fees>
```


Make sure to clean up any remaining child processes, close and delete FIFOs, and free memory before exiting.

Code Description

To support your implementation, you need to provide a *succinct* answer to each of the questions in the file `README.md`. The word limit is 150 for each question.

1. Describe how your exchange works, using diagram(s) if necessary.
2. Describe your design decisions for the trader and how it's fault-tolerant and efficient.
3. Describe your tests and how to run them.

Exchange: Example Outputs

```
# cat products.txt
2
GPU
Router
```

1 trader, 1 order

Command:

```
./pe_exchange products.txt ./trader_a
```

Orders:

```
Trader 0: BUY 0 GPU 30 500
```

Standard out:


```
[PEX] Starting
[PEX] Trading 2 products: GPU Router
[PEX] Created FIFO /tmp/pe_exchange_0
[PEX] Created FIFO /tmp/pe_trader_0
[PEX] Starting trader 0 (./trader_a)
[PEX] Connected to /tmp/pe_exchange_0
[PEX] Connected to /tmp/pe_trader_0
[PEX] [T0] Parsing command: <BUY 0 GPU 30 500>
[PEX]   --ORDERBOOK--
[PEX]   Product: GPU; Buy levels: 1; Sell levels: 0
[PEX]           BUY 30 @ $500 (1 order)
[PEX]   Product: Router; Buy levels: 0; Sell levels: 0
[PEX]   --POSITIONS--
[PEX]   Trader 0: GPU 0 ($0), Router 0 ($0)
[PEX] Trader 0 disconnected
[PEX] Trading completed
[PEX] Exchange fees collected: $0
```

2 traders, 6 orders

Command:

```
./pe_exchange products.txt ./trader_a ./trader_b
```

Orders:

```
Trader 0: BUY 0 GPU 30 500
Trader 0: BUY 1 GPU 30 501
Trader 0: BUY 2 GPU 30 501
Trader 0: BUY 3 GPU 30 502
Trader 1: SELL 0 GPU 99 511
Trader 1: SELL 1 GPU 99 402
```

Standard out:

```
[PEX] Starting
[PEX] Trading 2 products: GPU Router
[PEX] Created FIFO /tmp/pe_exchange_0
[PEX] Created FIFO /tmp/pe_trader_0
[PEX] Starting trader 0 (./trader_a)
[PEX] Connected to /tmp/pe_exchange_0
[PEX] Connected to /tmp/pe_trader_0
[PEX] Created FIFO /tmp/pe_exchange_1
[PEX] Created FIFO /tmp/pe_trader_1
[PEX] Starting trader 1 (./trader_a)
[PEX] Connected to /tmp/pe_exchange_1
[PEX] Connected to /tmp/pe_trader_1
[PEX] [T0] Parsing command: <BUY 0 GPU 30 500>
[PEX]   --ORDERBOOK--
[PEX]   Product: GPU; Buy levels: 1; Sell levels: 0
[PEX]           BUY 30 @ $500 (1 order)
[PEX]   Product: Router; Buy levels: 0; Sell levels: 0
[PEX]   --POSITIONS--
[PEX]   Trader 0: GPU 0 ($0), Router 0 ($0)
[PEX]   Trader 1: GPU 0 ($0), Router 0 ($0)
[PEX] [T0] Parsing command: <BUY 1 GPU 30 501>
[PEX]   --ORDERBOOK--
[PEX]   Product: GPU; Buy levels: 2; Sell levels: 0
[PEX]           BUY 30 @ $501 (1 order)
[PEX]           BUY 30 @ $500 (1 order)
[PEX]   Product: Router; Buy levels: 0; Sell levels: 0
[PEX]   --POSITIONS--
[PEX]   Trader 0: GPU 0 ($0), Router 0 ($0)
[PEX]   Trader 1: GPU 0 ($0), Router 0 ($0)
[PEX] [T0] Parsing command: <BUY 2 GPU 30 501>
[PEX]   --ORDERBOOK--
[PEX]   Product: GPU; Buy levels: 2; Sell levels: 0
[PEX]           BUY 60 @ $501 (2 orders)
[PEX]           BUY 30 @ $500 (1 order)
[PEX]   Product: Router; Buy levels: 0; Sell levels: 0
[PEX]   --POSITIONS--
[PEX]   Trader 0: GPU 0 ($0), Router 0 ($0)
[PEX]   Trader 1: GPU 0 ($0), Router 0 ($0)
```

```
[PEX] [T0] Parsing command: <BUY 3 GPU 30 502>
[PEX]   --ORDERBOOK--
[PEX]   Product: GPU; Buy levels: 3; Sell levels: 0
[PEX]           BUY 30 @ $502 (1 order)
[PEX]           BUY 60 @ $501 (2 orders)
[PEX]           BUY 30 @ $500 (1 order)
[PEX]   Product: Router; Buy levels: 0; Sell levels: 0
[PEX]   --POSITIONS--
[PEX]   Trader 0: GPU 0 ($0), Router 0 ($0)
[PEX]   Trader 1: GPU 0 ($0), Router 0 ($0)
[PEX] [T1] Parsing command: <SELL 0 GPU 99 511>
[PEX]   --ORDERBOOK--
[PEX]   Product: GPU; Buy levels: 3; Sell levels: 1
[PEX]           SELL 99 @ $511 (1 order)
[PEX]           BUY 30 @ $502 (1 order)
[PEX]           BUY 60 @ $501 (2 orders)
[PEX]           BUY 30 @ $500 (1 order)
[PEX]   Product: Router; Buy levels: 0; Sell levels: 0
[PEX]   --POSITIONS--
[PEX]   Trader 0: GPU 0 ($0), Router 0 ($0)
[PEX]   Trader 1: GPU 0 ($0), Router 0 ($0)
[PEX] [T1] Parsing command: <SELL 1 GPU 99 402>
[PEX] Match: Order 3 [T0], New Order 1 [T1], value: $15060, fee: $151.
[PEX] Match: Order 1 [T0], New Order 1 [T1], value: $15030, fee: $150.
[PEX] Match: Order 2 [T0], New Order 1 [T1], value: $15030, fee: $150.
[PEX] Match: Order 0 [T0], New Order 1 [T1], value: $4500, fee: $45.
[PEX]   --ORDERBOOK--
[PEX]   Product: GPU; Buy levels: 1; Sell levels: 1
[PEX]           SELL 99 @ $511 (1 order)
[PEX]           BUY 21 @ $500 (1 order)
[PEX]   Product: Router; Buy levels: 0; Sell levels: 0
[PEX]   --POSITIONS--
[PEX]   Trader 0: GPU 99 ($-49620), Router 0 ($0)
[PEX]   Trader 1: GPU -99 ($49124), Router 0 ($0)
[PEX] Trader 0 disconnected
[PEX] Trader 1 disconnected
[PEX] Trading completed
[PEX] Exchange fees collected: $496
```

Working on your assignment

Staff may make announcements on Ed (https://edstem.org/) regarding any updates or clarifications to the assignment. The Ed resources section will contain a PDF outlining any notes/changes/corrections to the assignment. You can ask questions on Ed using the assignments category. Please read this assignment description carefully before asking questions. Please ensure that your work is your own and you do not share any code or solutions with other students.

You can work on this assignment using your own computers or lab machines. You will need to submit to Ed via Git, which is covered elsewhere in this course. When you make a submission, your submission will be automatically compiled and run and you will receive feedback as to whether you passed the public test cases as well as a link that will enable you to inspect the output of your submission.

It is important that you continually back up your assignment files. You are encouraged to submit your assignment while you are in the process of completing it to receive feedback and to check for correctness of your solution.

Compilation and Execution

Your program will be compiled by running the default rule of a make file. Upon compiling your program should produce a two binaries: `pe_exchange`, `pe_trader`


```
make
./pe_exchange products.txt ./trader_a ./trader_b
```


Please make sure the above commands will compile and run your program. An example Makefile has been provided in the Scaffold, but you're encouraged to customize it to your needs. Additionally, consider implementing the project using multiple C source files and utilizing header files.

Tests will be compiled and run using two make rules; `make tests` and `make run_tests`.

```
make tests
make run_tests
```

These rules should build any tests you need, then execute each test and report back on your correctness.

Failing to adhere to these conventions will prevent your markers from running your code and tests. In this circumstance you will be awarded a mark of 0 for this assignment.

You are encouraged to submit multiple times, but only your last submission will be marked.

Writing your own test cases

We have provided you with some test cases but these do not test all the functionality described in the assignment. It is important that you thoroughly test your code by writing your own test cases, including both end-to-end (input / output) tests and unit tests using cmocka.

You should place all of your end-to-end test cases in the `tests/E2E` directory. Ensure that each test case has a .out output file and optionally an .in file. We recommend that the names of your test cases are descriptive so that you know what each is testing, e.g. buy-order.out and amend-order.out.

You should place all of your unit tests in the `tests` directory, under `unit-tests.c` (and more files, if necessary). All unit tests must be constructed with the cmocka unit testing framework.

You should have a brief description of your tests, and how they can be run, in `README.md`.

Error handling

Your code should have appropriate mechanisms in place to detect and handle errors, including but not limited to:

- NULL pointers
- Buffer overflows and underflows
- Unable to open FIFOs
- Unable to launch child processes
- Invalid or malformed commands

Restrictions

If your program does not adhere to these restrictions, your submission will receive 0.

- No Variable Length Arrays (VLAs)
- No threads (processes only)
- No excessive / 100% CPU usage when idling (eg: busy-waiting loops, active polling)
- Traders must be launched as child processes of the exchange process.

Marking

A grade for this assignment is made where there is a submission that compiles and the oral examination has been completed.

- Milestone Automated Test Cases - 5 - Passing automatic test cases.
- Final Automated Test Cases - 10 - Passing automatic test cases, a number of tests will *not* be released or run until after your final submission.

- Viva - 30 - You will need to answer questions from a COMP2017 teaching staff member regarding your implementation. You will be required to attend a zoom session with COMP2017 teaching staff member after the code submission deadline. A reasonable attempt will need to be made, otherwise you will receive zero for the assessment.

  To be eligible for Viva, you need to complete all test cases, whose name contains the keyword `VIVA`, before the final due date. These cases may be found in both milestone and final assignment.

  In this session, you will be asked to explain:

  - General questions about your understanding of the concepts needed for this assignment,
  - How you have arranged your memory, data structures and types
  - How you managed IPC (Inter Process Communication)
  - How you handle errors / make your code fault-tolerant
  - Answer further questions

Additionally marks will be deducted if:

- your program has memory leaks.
- excessive memory is used, e.g preallocation or over allocation of what is required.

- your program avoids *malloc* family of C functions to instantiate memory dynamically.
- your program does not use named pipes or signals for IPC.

- uses unjustifiable magic numbers.
- uses files, or mapped memory.
- all documentation parts of your submission, including code comments, README files, and so on, are not written in English.
- poor code readability will result in the deduction of marks. Your code and test cases should be neatly divided between header and source files in appropriate directories, should be commented, contain meaningful variable names, useful indentation, white space and functions should be used appropriately. Please refer to this course's style guide for more details.

Warning: Any attempts to deceive the automatic marking will result in an immediate zero for the entire assignment. Negative marks can be assigned if you do not properly follow the assignment description, or your code is unnecessarily or deliberately obfuscated.

Working on your assignment

You can work on this assignment on your own computer or the lab machines. It is important that you continually back up your assignment files onto your own machine, external drives, and in the cloud. You are responsible for your assignment files to remain private to you, and not accessible by others.

You are encouraged to submit your assignment on Ed while you are in the process of completing it. By submitting you will obtain some feedback of your progress on the sample test cases provided.

If you have any questions about any C functions, then refer to the corresponding **`man`** pages. You can ask questions about this assignment on Ed. As with any assignment, make sure that your work is your own, and that you do not share your code or solutions with other students.

Where to start

Begin with implementing the auto-trader (Milestone):

- read the full specification and make sure you understand the communication protocol.
- confirm you can open FIFOs setup by the exchange process.
- create a trader with basic functionality.

Exchange.

- confirm you can setup FIFOs and launch child processes

- confirm your processes can communicate through FIFOs and signals
- implement a simple order book for BUY / SELL orders
- check that it can handle multiple orders, identical orders and overlapping orders

Next, implement AMEND / CANCEL commands and the ability to handle multiple traders.

Next, implement the auto-trader, paying special attention to fault-tolerance and error recovery.