Systems Programming (2024 Fall)
Programming Assignment 2

B12902110 呂承諺

1 I/O and file descriptors

The following table shows the file descriptors that a process uses to communicate to the outside, its parent, or its children.

Table 1: File descriptors and their usages

fd	Root	Non-root nodes
0	Input from outside (command)	Input from parent (command)
1	Output to outside (answer)	
2	Output to outside (debug messages)	
3	`/dev/null`	Output to parent (response)
4,5,...	Pipe I/O to and from children, or FIFOs	

In this project, we use the standard I/O functions from the C library, and open every file descriptor as a stream either with `fopen()` or `fdopen()`. Every stream is set to unbuffered mode to avoid issues.

2 Main logic

2.1 Initialization

When a node is created, i.e., `friend` is executed, it performs the following initialization procedures.

1. Sets global variables `current_info`, `current_name`, `current_value`, and `is_root` according to `argv[1]`.
2. Opens `parent_write_stream` (corresponds to fd 3) for writing response to parent. For the root note, this stream will write to `/dev/null`.
3. Initialize the `children` array with `InitializeFriend()`.

2.2 Command dispatching

Then the node keeps listening for commands on stdin (fd 0), which for the root is from the outside and for all other nodes is from the parent. Depending on whether the command's *target node* is the current node or not, we either handle the command or relay the command to our children, by calling the corresponding handler or relay function respectively. Note that some commands require a lot of stuff done at the root node; in that case, we call the special root handler for that command if we're at the root node.

Table 2: Handler and relay functions for each command

Command	**Handler function**	**Relay function**	**Special root handler function**
Public commands			
Meet	`HandleMeet()`	`RelayMeet()`	N/A
Check	`HandleCheck()`	`RelayCheck()`	N/A
Graduate	`HandleGraduate()`	`RelayGraduate()`	A few lines in `main()`
Adopt	`HandleAdopt()`	`RelayAdopt()`	`RootHandleAdopt()`
Compare	`HandleCompare()`	`RelayCompare()`	`RootHandleCompare()`
Internal commands			
LevelPrint	`HandleLevelPrint()`	`RelayPrint()`	N/A
Search	`HandleSearch()`	`RelaySearch()`	N/A
AdoptPrint	`HandleAdoptPrint()`	`RelayAdoptPrint()`	N/A
CompareMod	`HandleCompareMod()`	N/A	N/A

Table 3: Condition for determining whether to handle the command

Command	**Condition for handling**
Public commands	
Meet	`current_name == parent_friend_name`
Check	`current_name == parent_friend_name`
Graduate	`current_name == friend_name`
Adopt	`is_root` / `current_name == parent_friend_name`
Compare	`is_root` / `current_name == friend_name`
Internal commands	
LevelPrint	`level == 0`
Search	`current_name == parent_friend_name`
AdoptPrint	`current_name == child_friend_name`
CompareMod	Always (No need to relay)

The following figure illustrates the dependency relationships between each command.

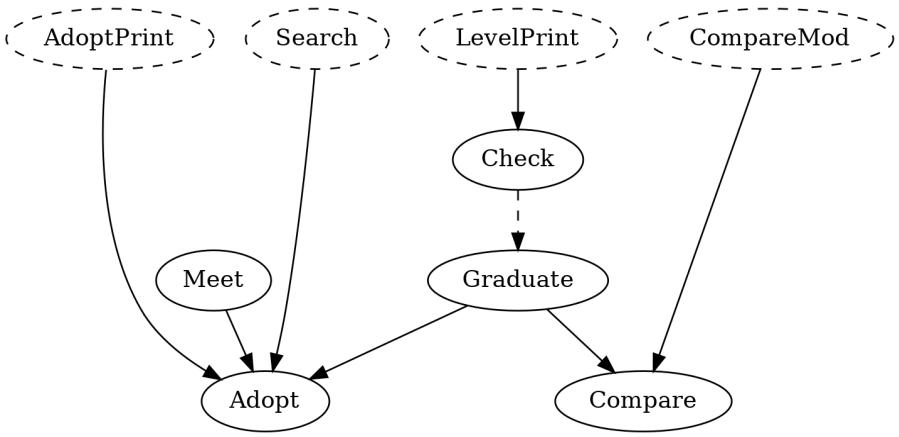

Figure 1: Dependency relationships between each command

2.3 Interprocess communication

The key idea is to use interprocess communication (IPC) to simulate depth-first search (DFS) and pre-order traversal on the tree. Sending a command to a child is like calling a function, and sending a response back to the parent is like returning from a function. DFS is crucial for this project, as it is used by all of the relay functions and some of the handler functions.

The table below shows the possible response codes that a node could send to its parent. They are implemented as type `response_t`.

Table 4: Response codes

Response code	Description
`REPSONSE_HANDLE_OK`	The current node has successfully handled the command.
`RESPONSE_RELAY_OK`	The current node has successfully relayed the command. That is, one of the node's descendants has successfully handled the command.
`RESPONSE_RELAY_OK_NO_PRINT`	Only used in the LevelPrint command. A special instance of `RESPONSE_RELAY_OK`, indicting nothing is printed at the target level for the subtree rooted at the current node. We need this to handle spaces and newlines for the Check command.
`RESPONSE_NOT_FOUND`	The target node for handling the command is not in the subtree of the current node, so we have to keep DFS-ing for the target node.
`RESPONSE_SEARCH_FOUND`	Only used in the Search command. The search target is found in the subtree of the current node.
`RESPONSE_SEARCH_FOUND`	Only used in the Search command. The search target is not found in the subtree of the current node.
`RESPONSE_EMPTY`	An error has occurred.

For the *relay* family of functions, their primary purpose is to DFS for the *target node* who is responsible for handling the command. Therefore, they send the command (mostly) as is to its children one at a time, and then wait for a response. If the response is either `RESPONSE_HANDLE_OK` or `RESPONSE_RELAY_OK`, we respond `RESPONSE_RELAY_OK` to the parent and end the search. This forms a relay chain back to the root node. Otherwise, we continue searching.

Some commands requires additional work to be done in the relay family of functions, such as Graduate, AdoptPrint, and CompareMod. For Graudate, it's because the target's parent need to wait for the target to die; for AdoptPrint and CompareMod, it's because they need access to the `children` array or name of the target's parent.

On the other hand, the *handler* family of functions are those who actually access the tree. Some handler functions also recursively dispatch the command to descendants and gather the results.

The return value for the handler and relay functions will be returned to `main()`, which subsequently will be the response code sent to the parent. Note that the root node has fd 3 opened at `/dev/null`; this means that any response the root node tries to send to its parent will be discarded, as there is no parent node by definition.

However, there are two special scenarios that doesn't use this mechanism:

1. Due to blocking I/O constraints regarding FIFOs, the Compare command requires responses be sent early in the handler function and the response mechanism in `main()` be suppressed. Here we use the `close()` of FIFOs as the response mechanism.

   For more information, see the implementation of `HandleAdopt()` and `HandleCompare()`.

2. For the handler part of Graduate, we use `waitpid()` as the response mechanism.

In conclusion, a command sent to a child is always paired with a response from that child, except Graduate. The response code is analogous to the return status of a function call.

3 Custom commands

Now we explain the usages and functionalities of custom commands.

3.1 Internal commands

The internal commands should only be invoked from nodes and never from the outside.

Table 5: Internal commands

Command	Usage	Description
LevelPrint	`L` *`LEVEL HAS_PRINTED`*	Print all nodes that are *`LEVEL`* levels deeper than the current node. *`HAS_PRINTED`* is a flag denoting whether anything about the target level has been printed.
Search	`S` *`PARENT CHILD`*	Search for *`CHILD`* under the subtree of *`PARENT`*. Essentially Check without printing.
AdoptPrint	`P` *`CHILD`*	Recursively print all parent-child relationships in the current subtree to file `Adopt.fifo`.
CompareMod	`%`	Recursively let all descendants have their `(friend_value * 2) % 99`.

3.2 Extensions to public commands

3.2.1 Meet

There are two variations of the Meet command, `Meet` and `meet`. `Meet` follows the description in the problem spec. `meet` is exactly the same as `Meet`, except that it suppress output, which is used in the Adopt command for creating new nodes. For more information, see `HandleMeet()` and `HandleAdopt()`.

3.2.2 Adopt

There are two variations of the Adopt command, `Adopt` and `adopt`. `adopt`, with a lowercase `a`, only opens the read end of `Adopt.fifo` in nonblocking mode, so that the AdoptPrint command can open the write end of `Adopt.fifo`. After all the information needed for adoption is in the FIFO, `Adopt`, with an uppercase `A`, is the one that actually reads the information and issues the Meet commands.

The adopt relay and handler functions accept a parameter `op` of type `adopt_op_t`, representing either one of "open mode" or "read mode". For more information, see `HandleAdopt()`.

4 Command implementations

4.1 Meet

4.1.1 Relay

- **Command passing**: As is.
- **Output**: If the target node is not found, output the corresponding answer.

4.1.2 Handler

The parent friend for Meet does the following operations.

1. Prepare two pipes to communicate with it, one for reads and one for writes.
2. `fork()` a new process and `exec() ./friend <child_friend_info>`.
3. Add an `Friend` entry in the `children` array.
4. Output answer response to outside.

4.2 Check

4.2.1 Relay

- **Command passing**: As is.
- **Output**: If the target node is not found, output the corresponding answer.

4.2.2 Handler

The root node of the subtree to be printed does the following operations.

1. Print the current node's information.
2. Run the LevelPrint command, which tells nodes that are 1 to `MAX_TREE_DEPTH` levels deeper than the current node to print their information, one level at a time, separated by a newline. In fact, this is iterative deepening depth-first search (IDFFS).

4.3 Graduate

4.3.1 Root handler (a few lines in `main()`)

Run Check first. If the target is not found, output the corresponding answer and end this command.

4.3.2 Relay

- **Command passing**: As is.
- **Output**: No output.
- **Special work when child is target**: Wait for it to die, and clean up it's entry in the `children` array, including closing pipe file descriptors that were used to communicate with it.
- **Special response mechanism**: If the child node is the target node, use `waitpid()` as the response mechanism. Otherwise use the normal response codes.

4.3.3 Handler

The root node of the subtree to be removed does the following operations.

1. Recursively send the Graduate command to each child. and wait for them to die with `waitpid()`. A child is dead implies its whole subtree is dead.

   Special response mechanism: Wait for a child to die with `waitpid()`.

2. Terminate the current process with `_exit()`.

4.4 Adopt

This command's target node is the adopting parent. In contrast, the target node of the AdoptPrint command is the root node of the subtree to be aodpted.

4.4.1 Root handler (a few lines in `main()`)

At the root node, do the following operations.

1. Run "`Search child_friend_name parent_friend_name`" to determine if the subtree to be adopted is a descendant of the adopting node. If yes, output the corresponding answer and end this command.
2. Create `Adopt.fifo` with `mkfifo()`.
3. Run the Adopt command in `ADOPT_OPEN` mode. Let the adopting node
4. Run the AdoptPrint command. This will print every parent-child information about the subtree to be adopted into the FIFO.
5. Graduate the original subtree that is about to be adopted.
6. Run the Adopt command in `ADOPT_READ` mode.
7. Remove the FIFO with `unlink()`.

4.4.2 Relay

- **Command passing**: As is.
- **Output**: No output.

4.4.3 Handler

The root node of the subtree to be adopted does the following operations.

`ADOPT_OPEN` **mode**: Open the read end of the FIFO in nonblocking mode.

`ADOPT_MODE` **mode**:

1. Read all contents in the FIFO and run Meet command for each parent-child relationship.
2. Close the FIFO.

4.5 Compare

4.5.1 Root handler

The root node does the following operations.

1. Create the two FIFOs with `mkfifo()`.
2. Pass down the compare command to the target node to be compared, so that it can open the read end of `number.fifo`.
3. Open the write end of `number.fifo` and write `number` to it.
4. Open the read end of `<friend_name>.fifo` and read the compare outcome.

   Special response mechanism: This step should block until the response can be read, so we use this as the response mechanism.
5. According to the compare outcome, output the corresponding answer message.

   If the compare outcome is `COMPARE_GRADUATE`, run "`Graduate friend_name`" to remove the subtree.
6. Close and remove the two FIFOs.

4.5.2 Relay

- **Command passing**: As is.
- **Output**: No output.
- **Special work when child is target**: Modify the value and info of the child's entry in the `children` array.

4.5.3 Handler

The target node that is compared with performs the following operations.

1. Early respond so that the root node can write the number to `number.fifo`.

   Special response mechanism: We use the blocking of `<friend_name>.fifo` as the response mechanism for this command.
2. Open the read end of `number.fifo` and read from it.
3. Compare the number read from `number.fifo` and write the compare outcome to `<friend_name>.fifo`.
4. Compare the number to the current node's value and write the output to the response FIFO.

   If the outcome is `COMPARE_MOD`, run CompareMod to have all descendants have their value modded.
5. Close the two FIFOs.

4.6 Internal commands

The implementation details for internal commands are omitted for brevity.