SO/SO1/OS Project Description

updated with Phase 2, and with Phase 2 hints

Build a C program in the UNIX environment simulating a treasure hunt game system that allows users to create, manage, and participate in digital treasure hunts. The system will use files to store treasure clues and locations, manage game state through processes and signals, and facilitate interaction between different components through pipes and redirects.

Each phase builds upon the previous one. You must present your program at each phase. Not implementing a phase disqualifies the entire project and the student will not get a passing grade.

Phase 1: File Systems (Weeks 6-7)

Goal: Create the foundation for storing and managing treasure hunt data using file operations.

Requirements:

1. Create a program called `treasure_manager` that:
    - Creates and reads treasure data files in a structured format
    - Uses files to store information about treasures storing at least the following fields:
        - Treasure ID
        - User name (unique, as text)
        - GPS coordinates (latitude and longitude as floating-point numbers)
        - Clue text (string)
        - Value (integer)
2. Implement the following operations, that the user can start by specifying appropriate options in the command line:
    - `add <hunt_id>`: Add a new treasure to the specified hunt (game session). Each hunt is stored in a separate directory.
    - `list <hunt_id>`: List all treasures in the specified hunt. *First print the hunt name, the (total) file size and last modification time of its treasure file(s), then list the treasures.*
    - `view <hunt_id> <id>`: View details of a specific treasure
    - `remove_treasure <hunt_id> <id>`: Remove a treasure
    - `remove_hunt <hunt_id>`: Remove an entire hunt

Treasures are stored in a file (optionally more files, see note below), belonging to the hunt directory. Multiple treasures must reside in a single file, as multiple structures containing the fields above. **Note: You can use multiple files, if you need for instance to group related treasures, e.g. keep all treasures of the same user in a single file. But this is not mandatory, and you still are required to store multiple records in a single file. Pay**

attention to operations such as `remove_treasure`, that *may* imply reorganizing the entire file (e.g., when a user removes a record in the middle of the file).

All operations done by the users of the program are logged (recorded) in the same directory as the hunt, in a special text file called `logged_hunt`.

Example of command line:

```
treasure_manager --remove 'game7' 'treasure2'
```

3. Use the following system calls for file operations:
    - `open()`, `close()`, `read()`, `write()`, `lseek()`
    - `stat()` or `lstat()` to retrieve file information
    - `mkdir()` to create directories if needed
4. Store data in a well-defined binary format:
    - Use fixed-size records for treasures
    - Include proper error handling for file operations
    - Implement basic data validation before writing to files
5. Create different types of files:
    - Regular files for treasure data.
    - Directories (folders) for organizing multiple treasure hunts. Each hunt resides in a different directory
    - Create in the root of the running program a **symbolic link** for each `logged_hunt` file in each hunt. The link name should be in the form `logged_hunt-<ID>`, where `<ID>` is the id of the respective hunt.

Deliverables for Phase 1:

- A working `treasure_manager` program that can manage records and hunts, logs the operations and creates appropriate symlinks, as described above.

Hints for Phase 1:

- Distinct hunts are stored in distinct directories. Note that there is no command for listing all the hunts, therefore you can implement the requirements without parsing directory structures
- Formats for Hunt IDs, and IDs in general, are mainly at your own choice. You can use, for example, as a Hunt ID the name of the directory, or a part of it that is unique (e.g., Hunt001, Hunt002, etc.)

Phase 2: Processes & Signals (Weeks 8-9)

Goal: Extend the treasure hunt system to use multiple processes and basic signal-based communication.

Requirements:

Write an additional program, called `treasure_hub` that presents a simple interactive interface for the user. The program understands the following commands given interactively:

- *start_monitor:* starts a separate background process that monitors the hunts and prints to the standard output information about them when asked to
- *list_hunts*: asks the monitor to list the hunts and the total number of treasures in each
- *list_treasures:* tells the monitor to show the information about all treasures in a hunt, the same way as the command line at the previous stage did
- *view_treasure:* tells the monitor to show the information about a treasure in hunt, the same way as the command line at the previous stage did
- *stop_monitor:* asks the monitor to end then returns to the prompt. Prints monitor's termination state when it ends.
- *exit:* if the monitor still runs, prints an error message, otherwise ends the program

If the *stop_monitor* command was given, any user attempts to input commands will end up with an error message until the monitor process actually ends. To test this feature, the monitor program will delay its exit (*usleep()*) when it responds. The communication with the monitor process is done using signals.

Note: For setting up signal behaviour, you must use *sigaction()*, not *signal()*.

Deliverables for Phase 2:

- A working `treasure_hub` program following all specifications above, and the `treasure_manager`, updated if needed.

Hints for Phase 2:

- When a command is given, you can send a signal to the monitor (*e.g.,* SIGUSR1)
- You can use files to communicate the command details to the monitor process
- You can detect when the monitor ends by using SIGCHLD

Rules for submitting the system programming lab assignments:

- each student must have a git repository and weekly commits
- each phase will end with a mandatory code submission in a special Milestone assignment on Campus Virtual (a phase is two weeks, as noted above). All submitted programs must work as required by the above specification
- failing to submit work for any of the phases **will void the entire project** (grade 2 at the lab for the project) (This means if you don't submit all phases you fail the project)