TASK MANAGER SYSTEM DOCUMENTATION

Abstract

This document outlines a console-based task management system implemented in C. The application's core functionality is to provide users with a persistent and interactive way to manage a list of tasks.

The program uses a **menu-driven interface** to enable fundamental operations, including adding, viewing, modifying, and deleting tasks. It incorporates **data persistence** by serializing and deserializing task data to a binary file, ensuring all user information is preserved between sessions. This approach demonstrates a practical application of C's low-level features for efficient resource management and concurrent programming.

Table of Contents

Program

The Task Manager System is a console application designed to help users manage a list of tasks. It supports basic CRUD (Create, Read, Update, Delete) operations and introduces advanced features such as data persistence (saving and loading tasks to a file), multi-threaded task execution, and various sorting and search options.

Functionality

- **Main Menu**: After loading, the program presents a menu with eight options, including adding, viewing, searching, deleting, modifying, sorting, and executing tasks.

- **User Interaction**: The user selects an option by entering a number. The program then executes the corresponding function.

- **Task Operations**:

    - **Adding a Task**: The user is prompted to input details like a description, priority, and duration. A unique ID is automatically assigned, and the task is added to the list.
    - **Modifying/Deleting**: For these actions, the user provides a task's ID to specify which task to change or remove.
    - **Sorting/Searching**: These functions organize or find tasks based on specific criteria like priority or keywords.

- **Data Persistence**: After any change to the task list (adding, modifying, or deleting), the program automatically saves the updated list to a binary file, ensuring the data is preserved.

- **Task Execution**: This is a more advanced feature where the program simulates a task's progress. It can "run" tasks one by one or concurrently using **multi-threading**, which allows multiple tasks to be processed at the same time. The program's core logic is to read user input, perform the requested action, and then save the changes.

Tutorial

Step 1: Save the Code

First, you need to save the entire C code into a file. Make sure the file extension is .c.

Step 2: Install a C Compiler

To run a C program, you need a compiler that translates the C code into an executable file that your computer can run.

- For Windows: The easiest way is to install MinGW-w64. You can download the installer from the MinGW-w64 website. Follow the installation wizard.

- For macOS: You can install Apple's Command Line Tools, which include the GCC compiler. Open the Terminal application and run xcode-select --install.
- For Linux: Most Linux distributions come with GCC pre-installed. You can check by typing gcc --version in your terminal. If it's not installed, you can get it using your package manager (e.g., sudo apt install build-essential for Debian/Ubuntu).

Step 3: Compile the Code

Once you have a compiler, you'll use a terminal or command prompt to compile your C file.

1. Open your terminal or command prompt.
2. Navigate to the directory where you saved task_manager.c using the cd command.
3. Type the following command and press Enter:

gcc task_manager.c -o task_manager -lpthread

If the compilation is successful, you won't see any output, and a new file named task_manager (or task_manager.exe on Windows) will be created in your directory.

Step 4: Run the Program

Now that you have the executable file, you can run the program.

1. In the same terminal window, type the following command and press Enter:

./task_manager

2. The program will start, and you will see the main menu displayed in your terminal.
3. You can then enter a number from the menu to perform different actions and test out all the features of your task manager. The program will continue to run until you enter 8 to exit.

C Standard Libraries

The program uses several standard C libraries to function correctly

- **`stdio.h`** This library provides standard input and output functions like `printf()` for printing to the console and `scanf()` for reading user input. It is fundamental for any console-based program.
- **`stdlib.h`** This library includes functions for general utilities, such as dynamic memory allocation (`malloc`, `free`), control over program flow (`exit`), and other helpful functions.
- **`string.h`** This library is used for manipulating strings. Functions like `strcpy()` for copying strings, `strlen()` for getting string length, and `strcmp()` for comparing strings are used extensively, especially for handling task descriptions.
- **`ctype.h`** This library provides functions for character handling, such as `isspace()` which is used in `clearInputBuffer()` to check for whitespace characters.

- **`pthread.h`** This is a crucial library for the multi-threading feature. It provides functions for creating, managing, and joining threads, enabling the program to execute multiple tasks concurrently.
- **`time.h`** This library is used for handling time and dates. The `time()` function is used to get the timestamp for a task's creation, and the `sleep()` function (or its platform-specific equivalent) is used to simulate task duration.
- **`stdbool.h`** This library allows the use of the `bool` data type, `true`, and `false` keywords, making the code more readable and expressive when handling boolean flags like a task's completion status.

Data Types

The code uses several standard C data types as well as custom-defined types:

- **Primitive Types**:
  - `int`: Used for task IDs, durations, counts (`taskCount`), and choices in the menu.
  - `char`: Used for characters, especially within string arrays like `description`.
  - `bool`: A boolean type from `stdbool.h` to represent `true` or `false` for a task's completion status.
  - `time_t`: An integer type from `time.h` to store the creation time of a task.
  - `FILE*`: A pointer to a file stream, used for reading and writing to files.
  - `pthread_t`: A data type from `pthread.h` to represent a thread.
- **Derived/User-Defined Types**:
  - `Task`: A `struct` that encapsulates all the properties of a single task.
  - `Priority`: An `enum` that defines the three priority levels (HIGH, MEDIUM, LOW).
  - `ThreadArgs`: A `struct` specifically created to pass multiple arguments to the thread function.
- **Pointers**
  - `Task*`: A pointer to a Task struct.
  - `int*`: A pointer to an integer.
  - `char*`: A pointer to a character (used to represent strings).
  - `void*`: A generic pointer type used to pass data to threads.

Global Constants and Variables

Constant/Variable	Value	Description
`MAX_TASKS`	`100`	The maximum number of tasks the system can store. This prevents a buffer overflow and defines the size of the `tasks` array.
`MAX_DESCRIPTION`	`256`	The maximum length of a task's description string. This is a buffer size to prevent a description from being too long.

`FILENAME`	`"tasks.dat"`	The name of the binary file used for data persistence. The binary format is chosen for efficiency in reading/writing the `Task` structs.
`MAX_SIMULTANEOUS_TASKS`	`10`	The maximum number of tasks that can be executed concurrently. This limits the number of threads created, which is important for system performance.
`tasks`	`Task[MAX_TASKS]`	An array of `Task` structs that acts as the in-memory database for all tasks. All operations are performed on this array before being saved to the file.
`taskCount`	`0`	A global counter that keeps track of the number of active tasks. It's essential for managing the `tasks` array and iterating over tasks.
`nextTaskId`	`1`	An auto-incrementing ID assigned to each new task. This ensures every task has a unique identifier, which is crucial for searching, modifying, and deleting tasks.

Data Structures

`Priority` (Enum)

This enumeration defines the priority levels for a task. The integer values are used to facilitate easy sorting (lower integer value means higher priority). This is a clean way to represent discrete states with meaningful names.

```
typedef enum {
    LOW = 5,
    MEDIUM = 3,
    HIGH = 1
} Priority;
```

`Task` (Struct)

This structure represents a single task in the system. It encapsulates all the necessary data for a task in a single, organized unit.

```
typedef struct {
    int id;
    char description[MAX_DESCRIPTION];
    Priority priority;
    int duration; // in seconds
    time_t created;
    bool completed;
} Task;
```

Field	Type	Description
`id`	`int`	A unique identifier for the task, used for quick retrieval and modification.
`description`	`char[MAX_DESCRIPTION]`	A brief description of the task, capturing its essence.
`priority`	`Priority`	The priority level of the task, allowing for easy sorting and prioritization of work.
`duration`	`int`	The simulated duration of the task in seconds, used in the `executeTasks` functions to provide a realistic execution experience.
`created`	`time_t`	The timestamp when the task was created, enabling sorting by creation date.
`completed`	`bool`	A flag indicating whether the task is completed (`true`) or not (`false`). This is essential for tracking progress.

`ThreadArgs` (Struct)

This structure is a helper used to pass multiple arguments to the `executeTaskThread` function. Since `pthread_create` can only pass a single `void*` argument, this struct is necessary to bundle all the required information.

```
typedef struct {
    Task *task;
    int *isRunning;
    int taskIndex;
} ThreadArgs;
```

Field	Type	Description
`task`	`Task*`	A pointer to the task to be executed. The thread needs access to this data to simulate the task.
`isRunning`	`int*`	A pointer to a shared flag that can be used to signal the thread to stop. This is a basic mechanism for thread control.
`taskIndex`	`int`	The index of the task in the global `tasks` array. This is used to directly update the task's status upon completion.

Function Breakdown

`main()`

This is the entry point of the program. It orchestrates the entire application flow.

It first calls `loadTasksFromFile()` to retrieve any previously saved tasks. Then, it enters a `do-while` loop that continuously displays the menu using `showMenu()`, reads the user's choice, and

executes the corresponding function via a `switch` statement. The loop continues until the user selects option 8 to exit. The use of a `do-while` loop ensures the menu is shown at least once.

It is the central control hub that initializes the program state and manages all user interactions. Without it, the program would have no defined start or way to respond to user input.

`showMenu()`

```
void showMenu() {
    printf("\n╔═══════════════════════════════════════╗\n");
    printf("║       TASK MANAGER SYSTEM             ║\n");
    printf("╠═══════════════════════════════════════╣\n");
    printf("║ 1. Add New Task                       ║\n");
    printf("║ 2. View All Tasks                     ║\n");
    printf("║ 3. Search Tasks                       ║\n");
    printf("║ 4. Delete Task                        ║\n");
    printf("║ 5. Modify Task                        ║\n");
    printf("║ 6. Sort Tasks                         ║\n");
    printf("║ 7. Execute Tasks                      ║\n");
    printf("║ 8. Exit                              ║\n");
    printf("╚═══════════════════════════════════════╝\n");
    printf("Enter your choice (1-8): ");
}
```

To present a clear, formatted menu of all available actions to the user.

It uses `printf()` statements to display the menu options with ASCII art borders for a clean and user-friendly console interface.

It provides a visual guide for the user, making the command-line application intuitive to use.

`clearInputBuffer()`

```
// Helper function to clear input buffer
void clearInputBuffer() {
    int c;
    while ((c = getchar()) != '\n' && c != EOF);
}
```

To prevent leftover characters in the input buffer from interfering with subsequent `scanf()` or `fgets()` calls.

After a `scanf()` call, this function reads and discards all remaining characters in the input buffer until a newline character (`\n`) is encountered or the end of the file is reached.

This is a common and necessary practice in C to prevent unexpected behavior where a `scanf()` for a number is followed by a `fgets()` for a string. The `fgets()` would immediately read the leftover newline character, skipping the user's intended input.

`saveTasksToFile()`

```c
// Save tasks to file
void saveTasksToFile() {
    FILE *file = fopen(FILENAME, "wb");

    if (file == NULL) {
        printf("Error: Cannot open file for writing.\n");
        return;
    }

    // Write the next task ID first
    fwrite(&nextTaskId, sizeof(int), 1, file);

    // Write the number of tasks
    fwrite(&taskCount, sizeof(int), 1, file);

    // Write each task
    fwrite(tasks, sizeof(Task), taskCount, file);

    fclose(file);
    printf("Tasks saved to %s\n", FILENAME);
}
```

To provide **data persistence** by saving the current task list to a file.

It opens the binary file `tasks.dat` in write mode (`"wb"`). It then writes the total number of tasks (`taskCount`), the next available task ID (`nextTaskId`), and the entire `tasks` array to the file. This process serializes the in-memory data to a file on disk.

Without this function, the task list would be lost every time the program is closed. It ensures that the user's data is saved for future sessions.

`loadTasksFromFile()`

```c
// Load tasks from file
void loadTasksFromFile() {
    FILE *file = fopen(FILENAME, "rb");

    if (file == NULL) {
        printf("No saved tasks found. Starting with empty task list.\n");
        return;
    }

    // Read the next task ID
    fread(&nextTaskId, sizeof(int), 1, file);

    // Read the number of tasks
    fread(&taskCount, sizeof(int), 1, file);

    if (taskCount > MAX_TASKS) {
        printf("Warning: File contains more tasks than maximum allowed. Loading only %d tasks.\n", MAX_TASKS);
        taskCount = MAX_TASKS;
    }

    // Read each task
    fread(tasks, sizeof(Task), taskCount, file);

    fclose(file);
    printf("Loaded %d tasks from %s\n", taskCount, FILENAME);
}
```

To load the saved task list from the file into memory upon program start.

It opens the `tasks.dat` file in read mode (`"rb"`). If the file exists, it reads the `taskCount`, `nextTaskId`, and the entire `tasks` array from the file, effectively restoring the previous state of the application. If the file does not exist, it prints a message and the system starts with an empty task list.

This is the other half of the data persistence feature. It enables the application to resume from where the user last left off, providing a seamless experience.

`priorityToString(Priority p)`

```c
// Convert priority enum to string
const char* priorityToString(Priority p) {
    switch (p) {
        case HIGH: return "High";
        case MEDIUM: return "Medium";
        case LOW: return "Low";
        default: return "Unknown";
    }
}
```

To convert a `Priority` enum value into a readable string for display.

It uses a `switch` statement to return a string literal ("High", "Medium", or "Low") based on the `Priority` enum value passed to it.

Since the `Priority` enum is stored as an integer, this function is essential for displaying human-readable priority levels to the user, improving the user interface.

`isDuplicateTask(const char* description, ...)`

```
// Function to check if a task with similar description and properties already exists
bool isDuplicateTask(const char* description, Priority priority, int duration) {
    for (int i = 0; i < taskCount; i++) {
        // Check for exact description match
        if (strcmp(tasks[i].description, description) == 0) {
            printf("\n⚠ Similar task already exists! ⚠\n");
            displayTaskDetails(tasks[i]);
            printf("Do you still want to add this task? (1=Yes, 0=No): ");
            int confirm;
            scanf("%d", &confirm);
            return confirm != 1;
        }
    }
    return false;
}
```

To check for duplicate task descriptions before a new task is added.

This function iterates through all existing tasks and uses `strstr()` to see if the new task's description is a substring of any existing task's description. It then prompts the user for confirmation if a potential duplicate is found.

This prevents the user from accidentally creating redundant tasks.

`displayTaskDetails(Task task)`

```
// Function to display a single task with details
void displayTaskDetails(Task task) {
    char timeStr[30];
    strftime(timeStr, sizeof(timeStr), "%Y-%m-%d %H:%M:%S", localtime(&task.created));

    printf("┌──────────────────────────────────────────────────────────────┐\n");
    printf("│ Task ID: %-52d │\n", task.id);
    printf("├──────────────────────────────────────────────────────────────┤\n");
    printf("│ Description: %-48s │\n", task.description);
    printf("│ Priority:    %-48s │\n", priorityToString(task.priority));
    printf("│ Duration:    %-2d seconds                                    │\n", task.duration);
    printf("│ Created:     %-48s │\n", timeStr);
    printf("│ Status:      %-48s │\n", task.completed ? "Completed" : "Pending");
    printf("└──────────────────────────────────────────────────────────────┘\n");
}
```

To display all the details of a single task in a formatted, easy-to-read manner.

It uses a series of `printf()` statements to print each field of the `Task` struct, including the formatted creation date using `ctime()`, and the priority string using `priorityToString()`.

This provides a detailed view for the user, especially after performing a search or when viewing a list of tasks.

The displayTaskDetails function currently takes a Task struct by value. This means a copy of the entire struct is made. To optimize this, you could modify the function to accept a pointer to the Task struct, like void displayTaskDetails(const Task* task). This would make the function more efficient as it would only pass a memory address (a pointer) rather than the entire struct.

`addTask()`

```
void addTask() {
    if (taskCount >= MAX_TASKS) {
        printf("Task limit reached.\n");
        return;
    }

    Task t;
    t.id = nextTaskId++;
    t.created = time(NULL);
    t.completed = false;

    printf("\n=== Adding New Task ===\n");
    printf("Task ID: %d (auto-assigned)\n", t.id);

    clearInputBuffer();
    printf("Enter description: ");
    fgets(t.description, MAX_DESCRIPTION, stdin);
    t.description[strcspn(t.description, "\n")] = 0;

    int priorityChoice;
    do {
        printf("Select priority:\n");
        printf("1. High\n");
        printf("2. Medium\n");
        printf("3. Low\n");
        printf("Choice: ");
        scanf("%d", &priorityChoice);

        switch (priorityChoice) {
            case 1: t.priority = HIGH; break;
            case 2: t.priority = MEDIUM; break;
            case 3: t.priority = LOW; break;
            default: printf("Invalid choice. Try again.\n");
        }
    } while (priorityChoice < 1 || priorityChoice > 3);

    do {
        printf("Enter duration (in seconds, 1-3600): ");
        scanf("%d", &t.duration);

        if (t.duration < 1 || t.duration > 3600) {
            printf("Invalid duration. Please enter a value between 1 and 3600 seconds.\n");
        }
    } while (t.duration < 1 || t.duration > 3600);

    // Check for duplicate tasks
    if (isDuplicateTask(t.description, t.priority, t.duration)) {
        return;
    }

    tasks[taskCount++] = t;
    printf("\nTask added successfully!\n");

    saveTasksToFile();
}
```

To add a new task to the system.

It prompts the user for the task's description, priority, and duration. It uses `fgets()` to read the description and handles `scanf()` for the numerical inputs, including clearing the input buffer. It then assigns a unique ID, gets the current time, and adds the new task to the `tasks` array. Finally, it calls `saveTasksToFile()` to persist the change.

This is a core function, fulfilling the "Create" part of the program's CRUD functionality.

`viewTasks()`

```c
// Function to display all tasks
void viewTasks() {
    if (taskCount == 0) {
        printf("\nNo tasks available.\n");
        return;
    }

    printf("\n=== Task List (%d tasks) ===\n", taskCount);
    printf("┌──────┬─────────────────────────────┬──────────┬──────────┬──────────┐\n");
    printf("│ ID   │ Description                 │ Priority │ Duration │ Status   │\n");
    printf("├──────┼─────────────────────────────┼──────────┼──────────┼──────────┤\n");

    for (int i = 0; i < taskCount; i++) {
        // Truncate description if too long for display
        char shortDesc[30];
        strncpy(shortDesc, tasks[i].description, 25);
        shortDesc[25] = '\0';
        if (strlen(tasks[i].description) > 25) {
            strcat(shortDesc, "...");
        }

        printf("│ %-3d │ %-27s │ %-8s │ %-8d │ %-8s │\n",
               tasks[i].id,
               shortDesc,
               priorityToString(tasks[i].priority),
               tasks[i].duration,
               tasks[i].completed ? "Done" : "Pending");
    }
    printf("└──────┴─────────────────────────────┴──────────┴──────────┴──────────┘\n");

    printf("\nEnter task ID for details or 0 to return: ");
    int id;
    scanf("%d", &id);

    if (id != 0) {
        for (int i = 0; i < taskCount; i++) {
            if (tasks[i].id == id) {
                displayTaskDetails(tasks[i]);
                return;
            }
        }
        printf("Task not found.\n");
    }
}
```

To display a summary of all tasks and allow the user to view detailed information.

It first prints a table with columns for ID, description, and status. It then iterates through the `tasks` array, populating the table. After showing the summary, it asks the user to enter a task ID to see more details, and then calls `displayTaskDetails()` for the selected task.

This function allows the user to easily see their entire task list and then dive deeper into the details of any specific task.

`searchTasks()`

```c
// Search for tasks
void searchTasks() {
    if (taskCount == 0) {
        printf("\nNo tasks available to search.\n");
        return;
    }

    printf("\n=== Search Tasks ===\n");
    printf("1. Search by keyword\n");
    printf("2. Search by priority\n");
    printf("3. Return to main menu\n");
    printf("Choice: ");

    int choice;
    scanf("%d", &choice);

    switch (choice) {
        case 1: {
            char keyword[50];
            clearInputBuffer();
            printf("Enter keyword: ");
            fgets(keyword, sizeof(keyword), stdin);
            keyword[strcspn(keyword, "\n")] = 0;

            printf("\n=== Search Results ===\n");
            int found = 0;

            for (int i = 0; i < taskCount; i++) {
                if (strstr(tasks[i].description, keyword) != NULL) {
                    displayTaskDetails(tasks[i]);
                    found++;
                }
            }

            if (found == 0) {
                printf("No tasks found matching '%s'\n", keyword);
            } else {
                printf("%d task(s) found.\n", found);
            }
            break;
        }
        case 2: {
            int priorityChoice;
            printf("Select priority to search for:\n");
            printf("1. High\n");
            printf("2. Medium\n");
            printf("3. Low\n");
            printf("Choice: ");
            scanf("%d", &priorityChoice);

            Priority searchPriority;
            switch (priorityChoice) {
                case 1: searchPriority = HIGH; break;
                case 2: searchPriority = MEDIUM; break;
                case 3: searchPriority = LOW; break;
                default: printf("Invalid choice.\n"); return;
            }

            printf("\n=== Search Results ===\n");
            int found = 0;

            for (int i = 0; i < taskCount; i++) {
                if (tasks[i].priority == searchPriority) {
                    displayTaskDetails(tasks[i]);
                    found++;
                }
            }

            if (found == 0) {
                printf("No tasks found with %s priority\n", priorityToString(searchPriority));
            } else {
                printf("%d task(s) found.\n", found);
            }
            break;
        }
        case 3:
            return;
        default:
            printf("Invalid choice.\n");
    }
}
```

To allow the user to find tasks based on specific criteria.

It offers two search modes by a keyword in the description or by priority. It prompts for the search term and then iterates through the task list, printing the details of any matching tasks.

As the task list grows, this function is crucial for quickly locating specific tasks without having to view the entire list.

`deleteTask()`

```c
// Delete a task
void deleteTask() {
    if (taskCount == 0) {
        printf("\nNo tasks available to delete.\n");
        return;
    }

    printf("\n=== Delete Task ===\n");

    // Display a compact list of tasks
    printf("Current tasks:\n");
    for (int i = 0; i < taskCount; i++) {
        printf("%d: %s (%s)\n",
               tasks[i].id,
               tasks[i].description,
               priorityToString(tasks[i].priority));
    }

    printf("\nEnter task ID to delete (or 0 to cancel): ");
    int id;
    scanf("%d", &id);

    if (id == 0) return;

    // Find and delete the task
    for (int i = 0; i < taskCount; i++) {
        if (tasks[i].id == id) {
            printf("Deleting task: %s\n", tasks[i].description);
            printf("Are you sure? (1=Yes, 0=No): ");
            int confirm;
            scanf("%d", &confirm);

            if (confirm == 1) {
                // Shift all tasks down to fill the gap
                for (int j = i; j < taskCount - 1; j++) {
                    tasks[j] = tasks[j + 1];
                }
                taskCount--;
                printf("Task deleted successfully.\n");
                saveTasksToFile();
            } else {
                printf("Deletion cancelled.\n");
            }
            return;
        }
    }

    printf("Task with ID %d not found.\n", id);
}
```

To remove a task from the list.

It prompts the user for a task ID to delete. After confirmation, it finds the task's index in the array. It then shifts all subsequent tasks one position up, effectively overwriting the deleted task, and decrements `taskCount`. It concludes by calling `saveTasksToFile()`.

This is the "Delete" part of the CRUD functionality, allowing users to remove tasks they no longer need.

`modifyTask()`

```c
// Modify a task
void modifyTask() {
    if (taskCount == 0) {
        printf("\nNo tasks available to modify.\n");
        return;
    }

    printf("\n=== Modify Task ===\n");

    // Display a compact list of tasks
    printf("Current tasks:\n");
    for (int i = 0; i < taskCount; i++) {
        printf("%d: %s (%s)\n",
               tasks[i].id,
               tasks[i].description,
               priorityToString(tasks[i].priority));
    }

    printf("\nEnter task ID to modify (or 0 to cancel): ");
    int id;
    scanf("%d", &id);

    if (id == 0) return;

    // Find the task
    for (int i = 0; i < taskCount; i++) {
        if (tasks[i].id == id) {
            Task *t = &tasks[i];

            printf("\n=== Modifying Task ID: %d ===\n", t->id);
            printf("1. Description: %s\n", t->description);
            printf("2. Priority: %s\n", priorityToString(t->priority));
            printf("3. Duration: %d seconds\n", t->duration);
            printf("4. Status: %s\n", t->completed ? "Completed" : "Pending");
            printf("5. Save and return\n");

            int choice;
            do {
                printf("\nSelect what to modify (1-5): ");
                scanf("%d", &choice);

                switch (choice) {
                    case 1:
                        clearInputBuffer();
                        printf("New description: ");
                        fgets(t->description, MAX_DESCRIPTION, stdin);
                        t->description[strcspn(t->description, "\n")] = 0;
                        break;
                    case 2: {
                        int priorityChoice;
                        printf("Select new priority:\n");
                        printf("1. High\n");
                        printf("2. Medium\n");
                        printf("3. Low\n");
                        printf("Choice: ");
                        scanf("%d", &priorityChoice);

                        switch (priorityChoice) {
                            case 1: t->priority = HIGH; break;
                            case 2: t->priority = MEDIUM; break;
                            case 3: t->priority = LOW; break;
                            default: printf("Invalid choice.\n");
                        }
                        break;
                    }
                    case 3:
                        do {
                            printf("New duration (1-3600 seconds): ");
                            scanf("%d", &t->duration);

                            if (t->duration < 1 || t->duration > 3600) {
                                printf("Invalid duration.\n");
                            }
                        } while (t->duration < 1 || t->duration > 3600);
                        break;
                    case 4:
                        t->completed = !t->completed;
                        printf("Status changed to: %s\n", t->completed ? "Completed" : "Pending");
                        break;
                    case 5:
                        printf("Changes saved.\n");
                        saveTasksToFile();
                        return;
                    default:
                        printf("Invalid choice.\n");
                }

            } while (choice != 5);

            return;
        }
    }

    printf("Task with ID %d not found.\n", id);
}
```

To update the details of an existing task.

It prompts the user for the ID of the task to modify. It then presents a sub-menu to choose which field to edit description, priority, duration, or completion status. It handles the input for the new value and updates the task in the array. Finally, it calls `saveTasksToFile()`.

This is the "Update" part of the CRUD functionality, providing flexibility for changing task details as needed.

`sortTasks()`

```c
// Sort tasks by priority and then duration
void sortTasks() {
    if (taskCount <= 1) {
        printf("\nNothing to sort.\n");
        return;
    }

    printf("\n=== Sort Tasks ===\n");
    printf("1. Sort by priority (highest first)\n");
    printf("2. Sort by duration (shortest first)\n");
    printf("3. Sort by creation time (newest first)\n");
    printf("Choice: ");

    int choice;
    scanf("%d", &choice);

    for (int i = 0; i < taskCount - 1; i++) {
        for (int j = i + 1; j < taskCount; j++) {
            bool shouldSwap = false;

            switch (choice) {
                case 1:
                    // By priority (then duration for ties)
                    shouldSwap = tasks[i].priority > tasks[j].priority ||
                                 (tasks[i].priority == tasks[j].priority &&
                                  tasks[i].duration > tasks[j].duration);
                    break;
                case 2:
                    // By duration (then priority for ties)
                    shouldSwap = tasks[i].duration > tasks[j].duration ||
                                 (tasks[i].duration == tasks[j].duration &&
                                  tasks[i].priority > tasks[j].priority);
                    break;
                case 3:
                    // By creation time (newer first)
                    shouldSwap = tasks[i].created < tasks[j].created;
                    break;
                default:
                    printf("Invalid choice. Nothing sorted.\n");
                    return;
            }

            if (shouldSwap) {
                Task temp = tasks[i];
                tasks[i] = tasks[j];
                tasks[j] = temp;
            }
        }
    }

    printf("Tasks sorted successfully.\n");
    viewTasks();
}
```

To organize the task list based on different criteria.

It uses `qsort()` from `stdlib.h` to sort the tasks. It provides different comparison functions (`compareByPriority`, `compareByDuration`, `compareByCreationTime`) that `qsort()` uses to determine the sorting order.

Sorting is an important utility for task management, allowing users to prioritize their work or see what's coming up next.

`executeTasks()`

```c
// Thread function to execute a single task
void* executeTaskThread(void* arg) {
    ThreadArgs* args = (ThreadArgs*)arg;
    Task* task = args->task;
    int* isRunning = args->isRunning;
    int taskIndex = args->taskIndex;

    printf("\n[Thread %d] Executing: %s (ID: %d) | Priority: %s | Duration: %d sec\n",
           taskIndex + 1, task->description, task->id,
           priorityToString(task->priority), task->duration);

    // Countdown timer
    for (int j = task->duration; j > 0; j--) {
        if (!(*isRunning)) {
            printf("\n[Thread %d] Task execution cancelled.\n", taskIndex + 1);
            pthread_exit(NULL);
        }

        printf("\r[Thread %d] Time remaining: %d seconds...   ", taskIndex + 1, j);
        fflush(stdout);
        sleep(1);  // Simulate execution
    }

    task->completed = true;
    printf("\r[Thread %d] Task %d completed!                \n", taskIndex + 1, task->id);

    free(args);
    pthread_exit(NULL);
}
```

To provide various ways to simulate task completion.

It presents a sub-menu with options to execute tasks sequentially, simultaneously, or a specific task. It then calls the appropriate function (`executeSpecificTask()`, `executeMultipleTasks()`, or a sequential execution loop) based on the user's choice.

This function provides the core "execution" logic and allows users to simulate the time and effort required for tasks.

`executeSpecificTask()`

```c
void executeMultipleTasks() {
    if (taskCount == 0) {
        printf("\nNo tasks to execute.\n");
        return;
    }
```

```
    // Count pending tasks
    int pendingCount = 0;
    int pendingTaskIds[MAX_TASKS];

    printf("\n=== Pending Tasks ===\n");

    for (int i = 0; i < taskCount; i++) {
        if (!tasks[i].completed) {
            pendingTaskIds[pendingCount++] = tasks[i].id;
            printf("%d: %s (%s, %d sec)\n",
                   tasks[i].id,
                   tasks[i].description,
                   priorityToString(tasks[i].priority),
                   tasks[i].duration);
        }
    }

    if (pendingCount == 0) {
        printf("No pending tasks to execute.\n");
        return;
    }

    // Task selection
    char taskSelection[MAX_TASKS * 4]; // Allow for IDs up to 999 with commas
    bool selectedTasks[MAX_TASKS] = {false};
    int numSelected = 0;

    printf("\nSelect tasks to execute (enter IDs separated by commas, or 'all'
for all tasks): ");
    clearInputBuffer();
    fgets(taskSelection, sizeof(taskSelection), stdin);

    // Remove newline
    taskSelection[strcspn(taskSelection, "\n")] = 0;

    // Check if user wants all tasks
    if (strcmp(taskSelection, "all") == 0) {
        for (int i = 0; i < pendingCount; i++) {
            for (int j = 0; j < taskCount; j++) {
                if (tasks[j].id == pendingTaskIds[i] && !tasks[j].completed) {
                    selectedTasks[j] = true;
                    numSelected++;
                }
            }
        }
```

```
    } else {
        // Parse comma-separated IDs
        char *token = strtok(taskSelection, " ,");
        while (token != NULL) {
            int id = atoi(token);
            if (id > 0) {
                for (int i = 0; i < taskCount; i++) {
                    if (tasks[i].id == id && !tasks[i].completed) {
                        selectedTasks[i] = true;
                        numSelected++;
                        break;
                    }
                }
            }
            token = strtok(NULL, " ,");
        }
    }

    if (numSelected == 0) {
        printf("No valid tasks selected.\n");
        return;
    }

    // Create threads (up to MAX_SIMULTANEOUS_TASKS)
    int maxThreads = (numSelected < MAX_SIMULTANEOUS_TASKS) ? numSelected :
MAX_SIMULTANEOUS_TASKS;
    pthread_t threads[MAX_SIMULTANEOUS_TASKS];
    int runningThreads = 0;
    int isRunning = 1; // Flag to control thread execution
    int tasksDone = 0;

    printf("\nExecuting %d tasks with up to %d running simultaneously.\n",
numSelected, maxThreads);
    printf("Press Enter to start execution or Ctrl+C to cancel...");
    getchar();

    time_t startTime = time(NULL);

    // Initialize and create threads
    for (int i = 0; i < taskCount && runningThreads < maxThreads; i++) {
        if (selectedTasks[i]) {
            ThreadArgs *args = (ThreadArgs*)malloc(sizeof(ThreadArgs));
            if (args == NULL) {
                printf("Memory allocation error\n");
                return;
```

```
            }

            args->task = &tasks[i];
            args->isRunning = &isRunning;
            args->taskIndex = runningThreads;

            pthread_create(&threads[runningThreads], NULL, executeTaskThread,
(void*)args);
            runningThreads++;
            tasksDone++;
        }
    }

    // Wait for threads to finish and start new ones as needed
    int nextTaskToRun = 0;
    while (tasksDone < numSelected) {
        // Find the next task to run
        for (int i = nextTaskToRun; i < taskCount; i++) {
            if (selectedTasks[i] && !tasks[i].completed) {
                nextTaskToRun = i + 1;
                break;
            }
        }

        // Wait for any thread to finish
        for (int i = 0; i < runningThreads; i++) {
            pthread_join(threads[i], NULL);
        }

        // Start new threads for remaining tasks
        runningThreads = 0;
        for (int i = nextTaskToRun; i < taskCount && runningThreads < maxThreads;
i++) {
            if (selectedTasks[i] && !tasks[i].completed) {
                ThreadArgs *args = (ThreadArgs*)malloc(sizeof(ThreadArgs));
                if (args == NULL) {
                    printf("Memory allocation error\n");
                    return;
                }

                args->task = &tasks[i];
                args->isRunning = &isRunning;
                args->taskIndex = runningThreads;
```

```
                pthread_create(&threads[runningThreads], NULL, executeTaskThread,
(void*)args);
                runningThreads++;
                tasksDone++;

                if (tasksDone >= numSelected) {
                    break;
                }
            }
        }
    }

    // Wait for any remaining threads
    for (int i = 0; i < runningThreads; i++) {
        pthread_join(threads[i], NULL);
    }

    time_t endTime = time(NULL);
    printf("\n=== Execution Summary ===\n");
    printf("Tasks completed: %d\n", numSelected);
    printf("Total wall clock time: %ld seconds\n", (endTime - startTime));

    saveTasksToFile();
}
```

To simulate the execution of a single task.

After the user selects a task, it uses a for loop to simulate a countdown based on the task's duration. It prints a progress update every second.

It provides a basic, single-threaded way to "run" a task and visually track its progress.

The code uses a static array Task tasks[MAX_TASKS], so there is no dynamic memory allocation. However, ThreadArgs uses a pointer to a Task to point to the correct task within the static array.

executeMultipleTasks()

```c
// Execute a specific task
void executeSpecificTask() {
    if (taskCount == 0) {
        printf("\nNo tasks available to execute.\n");
        return;
    }

    printf("\n=== Execute Specific Task ===\n");

    // Display only pending tasks
    printf("Pending tasks:\n");
    int pendingCount = 0;
    for (int i = 0; i < taskCount; i++) {
        if (!tasks[i].completed) {
            printf("%d: %s (%s, %d sec)\n",
                   tasks[i].id,
                   tasks[i].description,
                   priorityToString(tasks[i].priority),
                   tasks[i].duration);
            pendingCount++;
        }
    }

    if (pendingCount == 0) {
        printf("No pending tasks to execute.\n");
        return;
    }

    printf("\nEnter task ID to execute (or 0 to cancel): ");
    int id;
    scanf("%d", &id);

    if (id == 0) return;

    // Find and execute the task
    for (int i = 0; i < taskCount; i++) {
        if (tasks[i].id == id && !tasks[i].completed) {
            printf("\nExecuting: %s (ID: %d) | Priority: %s | Duration: %d sec\n",
                   tasks[i].description, tasks[i].id,
                   priorityToString(tasks[i].priority),
                   tasks[i].duration);

            printf("Press Enter to start execution...");
            clearInputBuffer();
            getchar();

            // Countdown timer
            for (int j = tasks[i].duration; j > 0; j--) {
                printf("\rTime remaining: %d seconds...  ", j);
                fflush(stdout);
                sleep(1);  // Simulate execution
            }

            tasks[i].completed = true;
            printf("\rTask %d completed!                \n", tasks[i].id);
            saveTasksToFile();
            return;
        } else if (tasks[i].id == id && tasks[i].completed) {
            printf("Task %d is already marked as completed.\n", id);
            return;
        }
    }

    printf("Task with ID %d not found.\n", id);
}
```

To simulate the concurrent execution of multiple tasks using multi-threading.

It prompts the user to select which tasks to run. It then creates a `pthread_t` array to hold the threads. It iterates through the selected tasks, creating a new thread for each one and passing a

`ThreadArgs` struct to the `executeTaskThread` function. After creating the threads, it joins them one by one, ensuring the main program waits for all tasks to complete before continuing.

This demonstrates a key advanced feature of the program the ability to handle concurrency, which is a powerful concept in modern software development.

executeTaskThread(void* arg)

```
// Thread function to execute a single task
void* executeTaskThread(void* arg) {
    ThreadArgs* args = (ThreadArgs*)arg;
    Task* task = args->task;
    int* isRunning = args->isRunning;
    int taskIndex = args->taskIndex;

    printf("\n[Thread %d] Executing: %s (ID: %d) | Priority: %s | Duration: %d sec\n",
           taskIndex + 1, task->description, task->id,
           priorityToString(task->priority), task->duration);

    // Countdown timer
    for (int j = task->duration; j > 0; j--) {
        if (!(*isRunning)) {
            printf("\n[Thread %d] Task execution cancelled.\n", taskIndex + 1);
            pthread_exit(NULL);
        }

        printf("\r[Thread %d] Time remaining: %d seconds...   ", taskIndex + 1, j);
        fflush(stdout);
        sleep(1);  // Simulate execution
    }

    task->completed = true;
    printf("\r[Thread %d] Task %d completed!                \n", taskIndex + 1, task->id);

    free(args);
    pthread_exit(NULL);
}
```

The function that runs within a separate thread to simulate a task.

It casts the `void*` argument back to a `ThreadArgs` struct to access the task data. It then simulates the duration using a `for` loop with a `sleep(1)` call. A progress indicator is printed to the console from within the thread. Upon completion, it marks the task as completed in the global `tasks` array.

This is the engine of the multi-threading feature. It encapsulates the logic for running a single task in its own thread, allowing for concurrent execution.

The executeTaskThread(void* arg) function uses a pointer (void* arg) to pass a ThreadArgs struct. This is a requirement of the pthread_create function. By passing a pointer, the function avoids copying the entire struct, which contains a pointer to a Task struct itself.

The executeTaskThread function uses args->task->completed = true to modify the completed field of the original Task struct in the global tasks array. Without the pointer, the thread would only be able to modify a local copy of the Task, and the change would not be reflected in the main program.