PHASE1.C

1. **Includes Libraries**: It includes necessary libraries for input/output operations, process management, and string handling.
2. **Defines Constants**: It defines constants for the maximum size of input and the maximum number of arguments.
3. **Prints a Prompt**: The **print\_prompt()** function displays a prompt (**phase1-shell>**) for the user to enter commands.
4. **Displays Help**: The **print\_help()** function lists the available commands that the shell can execute.
5. **Main Loop**:
   * The **main()** function contains an infinite loop that keeps the shell running until the user decides to exit.
   * It prompts the user for input and reads a line of text using **fgets()**.
   * It removes any newline character from the input.
   * It splits the input string into individual arguments (words) using **strtok()**.
6. **Handles Commands**:
   * If the user types **exit**, the shell prints "exiting..." and breaks the loop to terminate.
   * If the user types **help**, it calls the **print\_help()** function to display available commands.
7. **Forks a Process**:
   * The shell creates a new process using **fork()**. This allows it to run commands independently.
   * In the child process (where **pid == 0**), it attempts to execute the command using **execvp()**. If this fails, it prints an error message and exits.
   * In the parent process, it waits for the child process to finish using **wait()**.
8. **Exiting the Shell**: When the user types **exit**, the loop ends, and the program terminates.

In summary, this code creates a basic shell that can execute a limited set of commands, handle help requests, and allow for exiting the shell.

PHASE 2.C

1. **Includes and Definitions**:
   * The code includes standard libraries like **<stdio.h>**, **<stdlib.h>**, **<unistd.h>**, **<sys/types.h>**, **<sys/wait.h>**, and **<string.h>**.
   * It defines constants for maximum input size, maximum argument size, and maximum history size.
2. **Function Definitions**:
   * **print\_prompt()**: Displays the shell prompt (**phase2-shell>**).
   * **print\_help()**: Prints a list of available commands.
   * **print\_history()**: Displays the command history stored in an array.
   * **clear\_history()**: Clears the command history by resetting the count.
3. **Main Function**:
   * The main function contains the core loop of the shell that runs indefinitely until the user decides to exit.
   * It initializes variables for input, arguments, command history, and process management.

**Main Loop Functionality:**

1. **Prompt Display**:
   * The shell displays a prompt to the user using **print\_prompt()**.
2. **Input Handling**:
   * It reads a line of input from the user using **fgets()**. If reading fails, it prints an error message.
   * The newline character from the input is removed using **strcspn()**.
3. **Command History**:
   * The entered command is stored in the history array if the history count is below the maximum allowed size.
4. **Tokenization**:
   * The input command is tokenized into arguments using **strtok()**, which splits the input string by spaces.
5. **Command Handling**:
   * The shell checks for specific commands:
     + **exit**: Exits the shell.
     + **help**: Displays available commands.
     + **history**: Prints the command history.
     + **clearhistory**: Clears the command history.
6. **Process Creation**:
   * If the command is not one of the special commands, the shell forks a new process using **fork()**.
   * In the child process, it attempts to execute the command using **execvp()**. If execution fails, it prints an error and exits.
   * The parent process waits for the child to finish using **wait()**.

**Summary:**

This code provides a basic shell interface that allows users to execute system commands, view command history, and manage that history. It handles common shell tasks such as forking processes and executing commands while providing user-friendly features like help and history management.

PHASE 3.C

**Overview**

The code implements a simple command-line shell in C. A shell is a program that allows users to interact with the operating system by entering commands. This particular shell supports basic commands, command history, and multi-threading for executing commands concurrently.

**Key Components**

1. **Header Files**:
   * The code includes several standard libraries:
     + **<stdio.h>**: For input and output functions.
     + **<stdlib.h>**: For memory allocation and process control.
     + **<unistd.h>**: For POSIX operating system API (e.g., fork, exec).
     + **<sys/types.h>**, **<sys/wait.h>**: For handling process types and waiting for process termination.
     + **<string.h>**: For string manipulation functions.
     + **<pthread.h>**: For using threads.
2. **Constants**:
   * **MAX\_INPUT\_SIZE**: Maximum size for input commands (1024 characters).
   * **MAX\_ARG\_SIZE**: Maximum number of arguments per command (100).
   * **MAX\_HISTORY\_SIZE**: Maximum number of commands to store in history (100).
3. **Global Variables**:
   * **pthread\_mutex\_t history\_mutex**: A mutex for synchronizing access to the command history, ensuring thread safety.

**Function Definitions**

1. **print\_prompt()**:
   * Displays the shell prompt (**phase3-shell>**), indicating that the shell is ready for user input.
2. **print\_help()**:
   * Lists available commands that the user can execute in the shell.
3. **print\_history(char history[][MAX\_INPUT\_SIZE], int history\_count)**:
   * Prints the command history. It locks the mutex to ensure that no other thread modifies the history while it's being printed.
4. **clear\_history(char history[][MAX\_INPUT\_SIZE], int \*history\_count)**:
   * Clears the command history by resetting the count of stored commands to zero, ensuring that the history is empty.
5. **run\_command(void \*arg)**:
   * A function that runs a command in a separate thread. It takes a pointer to an array of arguments, attempts to execute the command using **execvp()**, and handles errors if execution fails.
6. **get\_command(char \*prompt, char \*input)**:
   * Displays a prompt and reads user input into the provided input buffer. It removes the newline character at the end of the input.

**Main Function**

The **main()** function contains the main loop of the shell:

1. **Initialization**:
   * Initializes the input buffer, argument array, command history, and history count.
   * Initializes the mutex for thread safety.
2. **Main Loop**:
   * Runs indefinitely until the user types **exit**.
   * Displays the prompt and reads user input.
   * Removes the newline character from the input.
   * Saves the command to history while ensuring thread safety with a mutex lock.
3. **Tokenizing Input**:
   * The input command is split into arguments using **strtok()**, which tokenizes the string based on spaces.
4. **Command Handling**:
   * Checks for specific commands:
     + **exit**: If the command is **exit**, the shell prints a message and breaks the loop to terminate.
     + **help**: Calls **print\_help()** to display available commands.
     + **history**: Calls **print\_history()** to show the command history.
     + **clearhistory**: Calls **clear\_history()** to clear the command history.
     + **join**: If the command is **join**, it prompts the user to enter two commands, tokenizes them, and executes them concurrently using threads.
5. **Forking a Process**:
   * If the command is not a special command, the shell forks a new process using **fork()**.
   * In the child process, it uses **execvp()** to execute the command. If it fails, it prints an error message and exits.
   * The parent process waits for the child process to finish using **wait()**.
6. **Cleanup**:
   * After exiting the main loop, the code destroys the mutex and returns from the **main()** function.

**Conclusion**

This code provides a basic but functional command-line shell that allows users to execute system commands, view their command history, and manage that history. It demonstrates fundamental concepts in C programming, such as process control, threading, and synchronization. The shell can be further enhanced by adding more features, such as input/output redirection, piping between commands, and handling more complex command syntax.