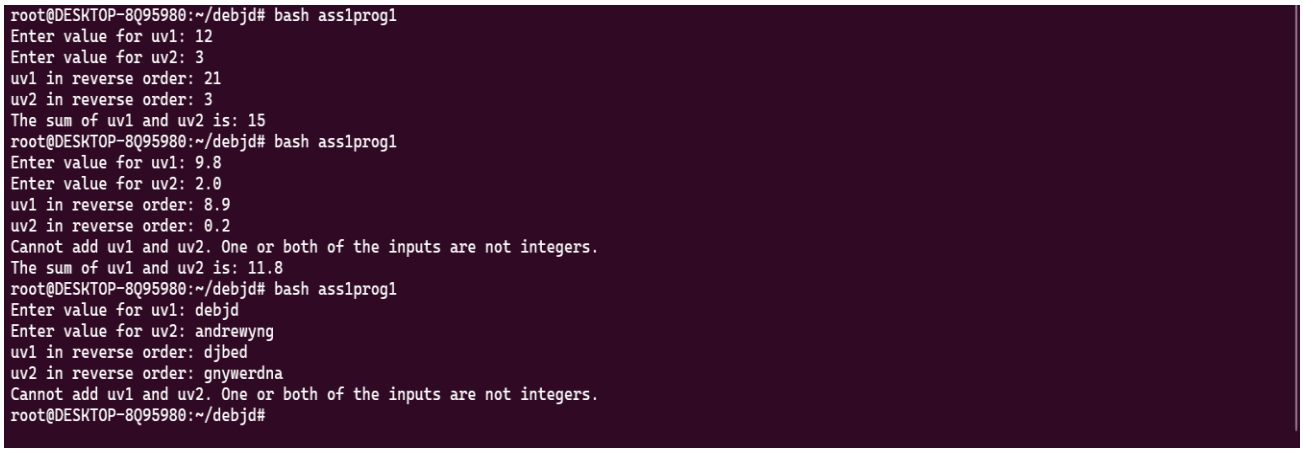
Explanantion:

This Bash script provides functionality to reverse strings and add numbers based on user input. It defines two functions: reverse_string, which uses the rev command to reverse the characters in a string, and add_numbers, which performs addition on two integer inputs using arithmetic expansion. The script prompts the user to enter values for uv1 and uv2. After receiving the inputs, it first prints the reversed versions of these strings. It then checks if both inputs are integers using a regular expression; if so, it computes their sum and displays the result. If either input is not an integer, an error message is shown. The script also checks if at least one of the inputs is a floating-point number using a different regular expression. If a float is detected, it attempts to add the two values using bc, a command-line calculator, and displays the result. If the addition fails due to invalid input, an appropriate error message is provided.

Output:



Explanation:

This Bash script interacts with the user to work with a file. It prompts the user to input a file name and then performs several operations depending on whether the file exists or not.

Here's a breakdown of its functionality:

1. **File Existence Check**: The script prompts the user to input a file name. It then checks if the file exists using the conditional -f "$file". If the file exists, it prints a message saying "It is a file." If the file does not exist, it displays "File not found" and creates a new text file with the specified name by writing the text "Create text file " into it.
2. **Counting Lines, Words, and Characters**:
    - The script then counts the number of lines in the file using wc -l, which is a command to get the line count, and stores the result in the variable count.
    - Similarly, it counts the number of words in the file using wc -w and stores it in the variable wcount.
    - It also counts the number of characters in the file using wc -c, saving the result in ccount.
3. **Displaying File Information**:
    - After counting the lines, words, and characters, it prints the corresponding numbers with descriptive messages.
    - Finally, it displays the content of the file using the cat command.

Output:

Explanation:

This Bash script counts the number of files and directories in the current working directory and its subdirectories. Here's a breakdown of what it does:

1. **Counting Files**:
    - The script uses the find command to locate all files in the current directory (.) and its subdirectories. The -type f option tells find to only look for files.
    - The output of find is then piped to wc -l, which counts the number of lines, i.e., the number of files found.
    - This value is stored in the variable file_count.
2. **Counting Directories**:
    - Similarly, the script uses find . -type d to locate all directories in the current directory and its subdirectories. The -type d option specifies that only directories should be included.
    - This output is also piped to wc -l to count the number of directories, and the result is stored in the variable dir_count.
3. **Displaying Results**:
    - Finally, the script prints the number of files and directories using echo, displaying the counts stored in file_count and dir_count.

Output:

Explanation:

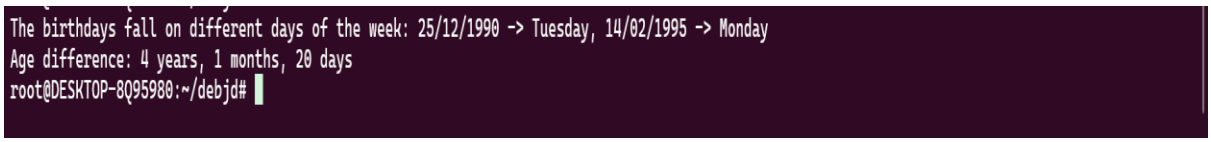
This Bash script calculates the difference in age between two birthdates and checks if the birthdates fall on the same day of the week. Here's a breakdown of its functionality:

1. **Helper Functions**:
    - **get_day_of_week**: This function takes a date as an argument and returns the day of the week (e.g., Monday, Tuesday) for that date using the date command.
    - **calculate_age_difference**: This function calculates the difference between two birthdates in years, months, and days. It does this by:
        - Converting each birthdate to a standard format (YYYYMMDD).
        - Calculating the difference in seconds using Unix timestamps (%s flag with date).
        - Converting the difference from seconds to days, and then computing the number of years, months, and days. The result is printed in a readable format, such as "X years, Y months, Z days."
2. **Input Validation**:
    - The script checks if exactly two arguments are provided (two birthdates in DD/MM/YYYY format). If not, it prints a usage message and exits.
3. **Processing the Birthdates**:
    - The birthdates are extracted from the script's arguments.
    - The day of the week for each birthdate is determined by calling the get_day_of_week function.
4. **Checking for Matching Days of the Week**:
    - The script compares the days of the week for the two birthdates. If they match, it prints a message indicating that both birthdays fall on the same day of the week. If they don’t

match, it displays both the birthdates and the corresponding days of the week.

5. **Calculating and Displaying the Age Difference**:
    - The script calls calculate_age_difference to compute the age gap between the two dates. The result, which includes the difference in years, months, and days, is then printed.

Output:



Explanation:

This Bash script allows the user to search for a specific word in a file and provides details on the total occurrences of the word, along with the line numbers where the word appears. Here's a breakdown of how the script works:

1. **Input Validation**:
    - The script first checks if exactly one argument (a filename) is provided when the script is run. If not, it displays a usage message and exits.
2. **File Existence and Readability Check**:
    - It verifies that the specified file exists and is readable using the -f and -r flags. If the file doesn't exist or is not readable, it prints an error message and exits.
3. **User Input for Word Search**:
    - The script prompts the user to input a word to search for in the file. This word is stored in the variable search_word.
4. **Searching for the Word**:
    - The script uses grep with the -o (only matching words) and -w (match whole words) options to search for occurrences of the word in the file. The total number of occurrences is then counted using wc -l.
5. **Handling Word Absence**:
    - If the word is not found in the file (i.e., the total occurrences are zero), the script displays a message stating that the word was not found and exits.
6. **Displaying Results**:
    - If the word is found, the script first displays the total number of occurrences of the word.
    - It then uses grep -n -o -w to print each line where the word occurs, combined with awk to count and display the number of occurrences for each line. The output shows the line numbers and the corresponding count of occurrences on each line.

Output:


root@DESKTOP-8Q95980:~/debjd# bash ass1prog6 t1.txt
Enter the word to search for: Salary
The word 'Salary' occurs 1 time(s) in the file.
Occurrences by line:
Line 1: 1 occurrence(s)
root@DESKTOP-8Q95980:~/debjd#

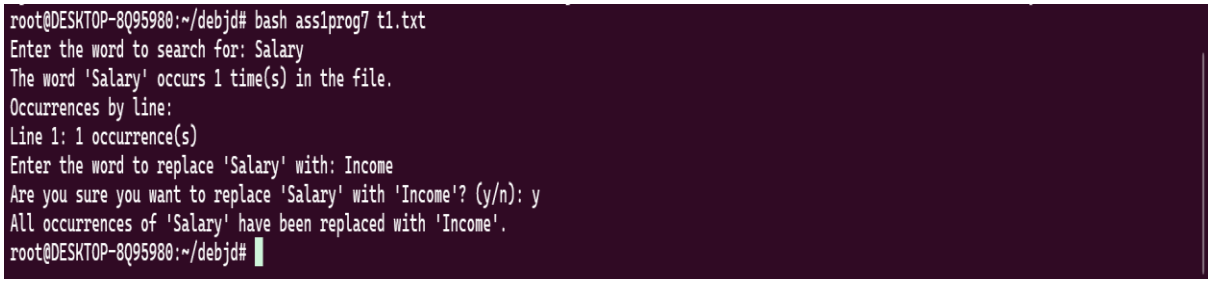
Explanation:

This Bash script allows the user to search for a specific word in a file, find its occurrences, and replace all instances of that word with another word provided by the user. Here's a detailed explanation of its functionality:

1. **Input Validation**:
    - The script checks if exactly one argument (a filename) is provided when the script is run. If not, it displays a usage message and exits.
2. **File Existence and Readability Check**:
    - It verifies that the provided file exists and is readable using the -f (file existence) and -r (readable file) flags. If the file does not exist or is not readable, an error message is shown, and the script exits.
3. **Word Search**:
    - The script prompts the user to input a word to search for in the file. It then uses grep -o -w to search for the word in the file:
        - -o: This option prints only the matched part of the word.
        - -w: This option ensures that only whole words are matched.
    - The total occurrences of the word in the file are counted using wc -l, and the result is displayed to the user.
4. **Display of Line-wise Occurrences**:
    - The script uses grep -n -o -w to find the occurrences of the word in the file along with the line numbers. This output is then passed to awk to count the number of occurrences for each line and print it in a structured format like "Line X: Y occurrence(s)."

5. **Replacement Prompt**:
    - After displaying the occurrences, the script prompts the user to input a second word, which will be used to replace the original search word. The user is asked to confirm whether they want to proceed with the replacement.
6. **Word Replacement**:
    - If the user confirms the replacement by typing y, the script uses the sed command to replace all occurrences of the search word with the new word:
        - \b: This ensures that the word is matched as a whole word (word boundary).
        - -i: This option modifies the file in-place.
    - If the replacement is successful, a message confirming the replacement is displayed. If an error occurs during the replacement, an error message is shown.

Output:



Explanation:

This C program implements a basic command-line shell. The shell accepts user input, parses the input into arguments, and executes built-in commands like cd, pwd, and echo, or forks a process to handle other external commands. Here's a detailed breakdown of the code:

Key Components:

1. **parse_command Function**:
    - This function takes the input string (command) and tokenizes it into individual arguments, using spaces or newline characters as delimiters.
    - strtok is used to split the input into tokens, and the tokens are stored in an array of strings (args). The last element of args is set to NULL to mark the end of the argument list, as required by the execvp function when executing external commands.
2. **execute_command Function**:
    - This function handles both built-in commands and external commands. It returns 1 to continue the shell or 0 to exit the shell.
    - **Built-in Commands**:
        - exit: Terminates the shell by returning 0.
        - cd: Changes the current working directory. If no argument is provided, an error message is printed.
        - pwd: Prints the current working directory.
        - echo: Prints the provided arguments to the console.
    - **External Commands**:
        - For non-built-in commands, the program forks a new process. The child process runs the command using execvp, which replaces the process image with the command to be executed.

- The parent process waits for the child process to finish using waitpid.

3. **Main Loop (main Function)**:
    - The program runs an infinite loop that acts as the shell prompt. It continuously displays the prompt (1234-shell>) and waits for user input.
    - The input is captured using fgets, which reads a line from standard input.
    - After capturing the input, the parse_command function splits it into arguments, and the execute_command function is called to execute the parsed command.
    - The loop continues until the user types exit to terminate the shell.

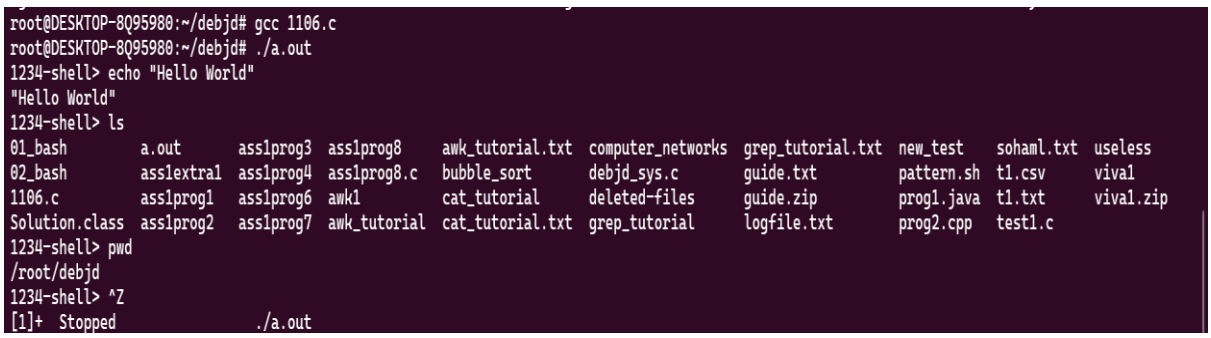
Handling External Commands:

- When a user types a command that is not built-in (e.g., ls, grep, etc.), the shell forks a new process and calls execvp, which replaces the current process image with the specified command. If the command is not found or an error occurs, the child process prints an error message using perror.

Error Handling:

- If cd is used without a valid argument, an error is displayed.
- If execvp fails (e.g., when trying to run a non-existent command), an error message is printed.
- The shell continues to run until the user types exit.

Output:



Explanation:

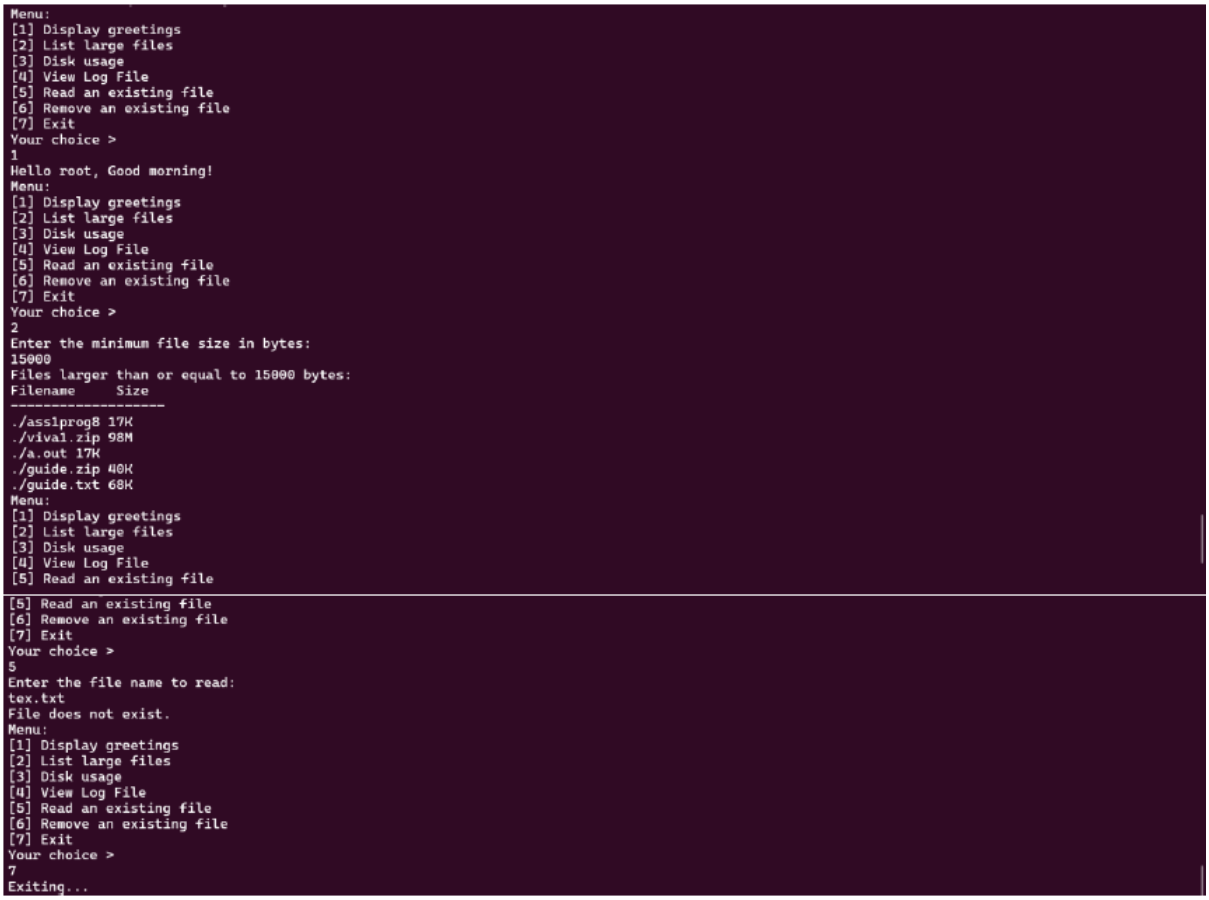
This is a simple implementation of a shell in C that can handle basic commands and perform operations like changing directories, displaying the current directory, echoing arguments, and running external commands. Here's a step-by-step breakdown:

Key Functionalities:

1. **Parsing the Command:**
    - The parse_command function tokenizes the user input (separated by spaces and newlines) and stores the resulting tokens in an array of strings (args). This array is passed to the execvp function to execute the command.
    - strtok is used to break the input into words. Each word is stored as an argument in args, and the array is terminated by NULL, as required by the execvp system call.
2. **Built-in Commands:**
    - The shell supports the following built-in commands:
        - exit: Ends the shell.
        - cd [directory]: Changes the current working directory to the one specified. If no argument is given, it prints an error message.
        - pwd: Prints the current working directory.
        - echo: Prints the arguments passed after echo.
3. **Executing External Commands:**
    - If the command is not a built-in one, the program forks a child process using fork. The child process then calls execvp to execute the external command (e.g., ls, grep, etc.).
    - The parent process waits for the child to finish using waitpid.
4. **Error Handling:**
    - The code includes error handling for several cases:
        - If chdir (used for changing directories) fails, it prints an error message.

- If execvp (used for external commands) fails, it prints an error message indicating the command couldn’t be executed.
- If fgets (used to read input) fails, it prints an error message and exits the shell.
- If the user enters an invalid command, the shell handles this gracefully by printing an error from execvp.

Output:



Program Extra 2: Add a system call (developed by you) to your linux shell.

Explanation:

Steps for Adding a Custom System Call

1 System Call Code

We will add a system call that returns the current number of processes running on the system. This will serve as a simple demonstration of kernel and shell interaction.

2 Modifying the Linux Kernel

Step 1: Modify System Call Table

- Navigate to the Linux kernel source directory, typically /usr/src/linux.
- Open the system call table file, arch/x86/entry/syscalls/syscall_64.tbl, and add the new system call definition:

Snippet:

```
440   64    my_syscall    sys_my_syscall
```

This assigns the system call number 440 to sys_my_syscall.

Step 2: Implement the System Call

- Create a C file to implement the system call, usually under kernel/. For example, create kernel/my_syscall.c with the following content:

Snippet:

```
#include <linux/kernel.h>

#include <linux/syscalls.h>

#include <linux/sched/signal.h>


SYSCALL_DEFINE0(my_syscall) {

    int process_count = 0;

    struct task_struct *task;
```

```
    for_each_process(task) {
        process_count++;
    }

    printk(KERN_INFO "Number of processes: %d\n", process_count);
    return process_count;
}
```

This code iterates through the task list and counts the total number of processes running in the system.

Step 3: Declare the System Call

- Declare the new system call by editing the include/linux/syscalls.h file:

Snippet:

```
asmlinkage long sys_my_syscall(void);
```

Step 4: Recompile the Kernel

- Compile and install the modified kernel:

Snippet:

```
make menuconfig  # Configure the kernel options (if needed)
make -j$(nproc)  # Compile the kernel
make modules_install
make install
```

- Reboot into the new kernel.

4.3 Testing the System Call

Once the kernel is recompiled and running, test the system call from user space.

1. Create a small C program to invoke the new system call:

Snippet:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/syscall.h>

int main() {
    long result = syscall(440);
    printf("Number of processes: %ld\n", result);
    return 0;
}
```

2. Compile and run this program:

Snippet:

```
gcc -o test_syscall test_syscall.c
./test_syscall
```

If the system call works, you should see the number of running processes printed to the terminal.

Adding the System Call to the Shell

1 Modifying the Shell

We can now modify the shell to include the new system call as part of its built-in commands.

- Open the shell source code (assuming you are developing your own shell or modifying an existing one).
- In the command parsing section, add a new command (e.g., proc_count) to invoke the system call:

Snippet:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/syscall.h>
```

```
void proc_count_command() {
    long result = syscall(440);
    printf("Number of processes running: %ld\n", result);
}

// Inside the shell command loop:
if (strcmp(command, "proc_count") == 0) {
    proc_count_command();
}
```

2 Rebuild the Shell

Recompile the shell to integrate the new functionality:

Snippet:

```
gcc -o my_shell my_shell.c
./my_shell
```

3 Testing the Shell Command

Once the shell is running, you can invoke the custom system call directly from the shell by typing:

Snippet:

```
proc_count
```

This should output the number of processes currently running on the system, leveraging the custom system call.

Output

1. System Call Test Output

After compiling and running the test program (test_syscall.c) that invokes the custom system call, the output would look like this:

```
$ ./test_syscall
Number of processes: 245
```

2. Custom Shell Output

After modifying the shell to include the proc_count command and recompiling it, the output would look like this when you run the shell and use the custom command:

```
$ ./my_shell
Welcome to My Shell! Type 'proc_count' to see the number of running processes.

> proc_count
Number of processes running: 245

> proc_count
Number of processes running: 247
```

Program Extra 2: Add a system call (developed by you) to your linux shell.

Explanation:

Steps for Adding a Custom System Call

1 System Call Code

We will add a system call that returns the current number of processes running on the system. This will serve as a simple demonstration of kernel and shell interaction.

2 Modifying the Linux Kernel

Step 1: Modify System Call Table

- Navigate to the Linux kernel source directory, typically /usr/src/linux.
- Open the system call table file, arch/x86/entry/syscalls/syscall_64.tbl, and add the new system call definition:

Snippet:

```
440   64    my_syscall    sys_my_syscall
```

This assigns the system call number 440 to sys_my_syscall.

Step 2: Implement the System Call

- Create a C file to implement the system call, usually under kernel/. For example, create kernel/my_syscall.c with the following content:

Snippet:

```
#include <linux/kernel.h>

#include <linux/syscalls.h>

#include <linux/sched/signal.h>


SYSCALL_DEFINE0(my_syscall) {

    int process_count = 0;

    struct task_struct *task;
```

```
    for_each_process(task) {
        process_count++;
    }

    printk(KERN_INFO "Number of processes: %d\n", process_count);
    return process_count;
}
```

This code iterates through the task list and counts the total number of processes running in the system.

Step 3: Declare the System Call

- Declare the new system call by editing the include/linux/syscalls.h file:

Snippet:

```
asmlinkage long sys_my_syscall(void);
```

Step 4: Recompile the Kernel

- Compile and install the modified kernel:

Snippet:

```
make menuconfig  # Configure the kernel options (if needed)
make -j$(nproc)  # Compile the kernel
make modules_install
make install
```

- Reboot into the new kernel.

4.3 Testing the System Call

Once the kernel is recompiled and running, test the system call from user space.

2. Create a small C program to invoke the new system call:

Snippet:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/syscall.h>

int main() {
    long result = syscall(440);
    printf("Number of processes: %ld\n", result);
    return 0;
}
```

3. Compile and run this program:

Snippet:

```
gcc -o test_syscall test_syscall.c
./test_syscall
```

If the system call works, you should see the number of running processes printed to the terminal.

Adding the System Call to the Shell

1 Modifying the Shell

We can now modify the shell to include the new system call as part of its built-in commands.

- Open the shell source code (assuming you are developing your own shell or modifying an existing one).
- In the command parsing section, add a new command (e.g., proc_count) to invoke the system call:

Snippet:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/syscall.h>
```

```
void proc_count_command() {
    long result = syscall(440);
    printf("Number of processes running: %ld\n", result);
}

// Inside the shell command loop:
if (strcmp(command, "proc_count") == 0) {
    proc_count_command();
}
```

2 Rebuild the Shell

Recompile the shell to integrate the new functionality:

Snippet:

```
gcc -o my_shell my_shell.c
./my_shell
```

3 Testing the Shell Command

Once the shell is running, you can invoke the custom system call directly from the shell by typing:

Snippet:

```
proc_count
```

This should output the number of processes currently running on the system, leveraging the custom system call.

Output

1. System Call Test Output

After compiling and running the test program (test_syscall.c) that invokes the custom system call, the output would look like this:

```
$ ./test_syscall
Number of processes: 245
```

2. Custom Shell Output

After modifying the shell to include the proc_count command and recompiling it, the output would look like this when you run the shell and use the custom command:

```
$ ./my_shell
Welcome to My Shell! Type 'proc_count' to see the number of running processes.

> proc_count
Number of processes running: 245

> proc_count
Number of processes running: 247
```