**LSP SIGNAL HANDLING DAY6**

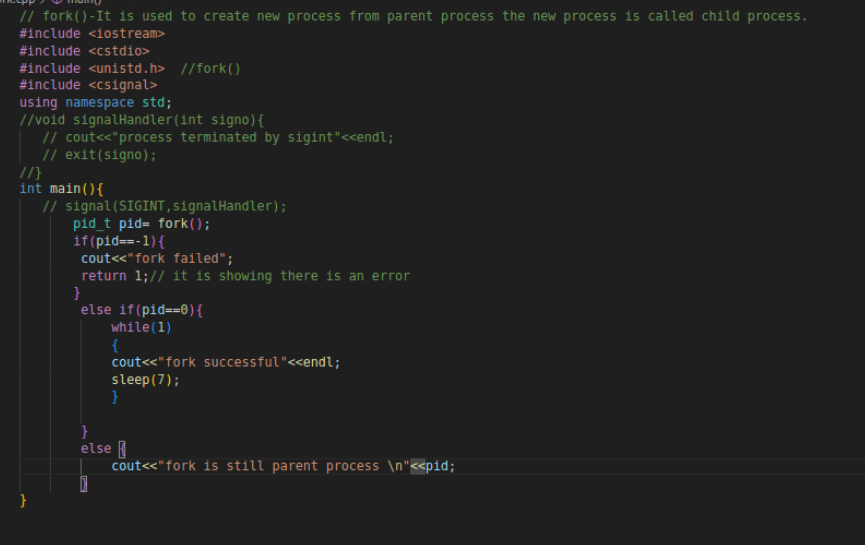
Child Process

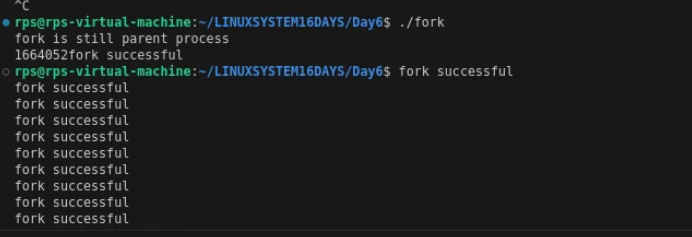
1. **Declaration**:
   * **pid\_t pid; declares a variable pid. Initially, it does not have any valid value related to process IDs.**
2. **After fork()**:
   * **Parent Process**:
     + **pid receives the PID of the child process. For example, if the child’s PID is 12345, then the pid in the parent will be 12345.**
   * **Child Process**:
     + pid is set to 0 in the child process.
3. **In Case of Failure**:
   * If fork() fails, pid will be a negative value, indicating an error.

### **Summary**

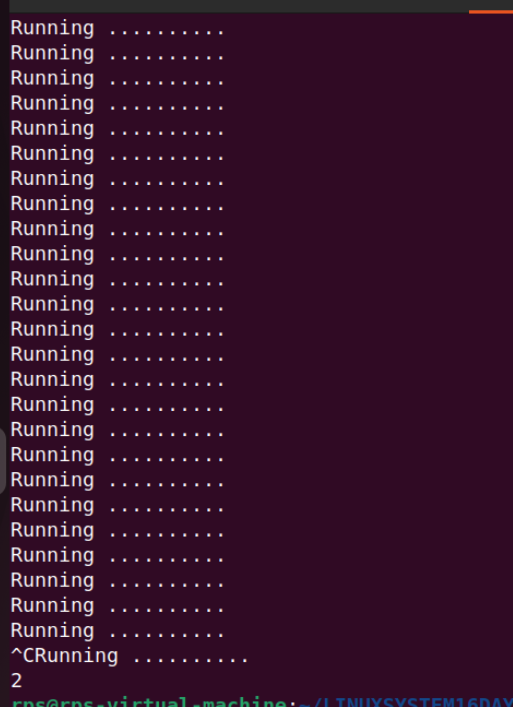
* **Before fork()**: The variable pid does not contain any valid process ID value.
* **After fork()**:
  + **Parent**: pid holds the PID of the newly created child process.
  + **Child**: pid is set to 0.
  + **Failure**: pid is a negative value indicating an error.

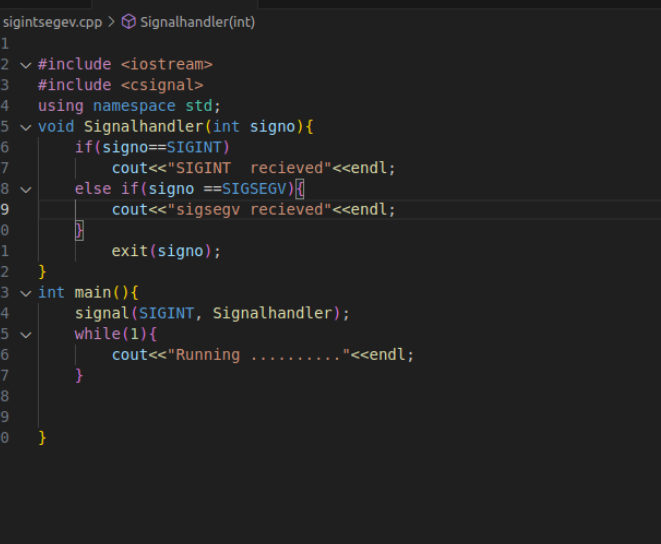
This mechanism allows the parent and child processes to differentiate their roles and manage their execution appropriately.





Basic Signal Handling  
Simple Signal Handler: Write a C++ program that handles the SIGINT signal (Ctrl+C) gracefully by printing a custom message before exiting.

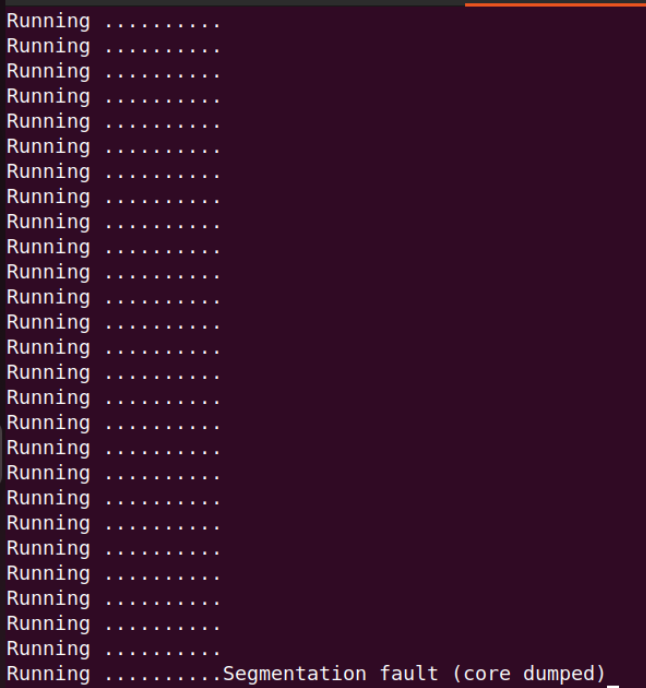


Multiple Signal Handling: Create a program that handles both SIGINT and SIGTERM signals, printing a different message for each.  
Ignoring Signals: Develop a program that ignores the SIGTERM signal and continues execution even after it's sent.

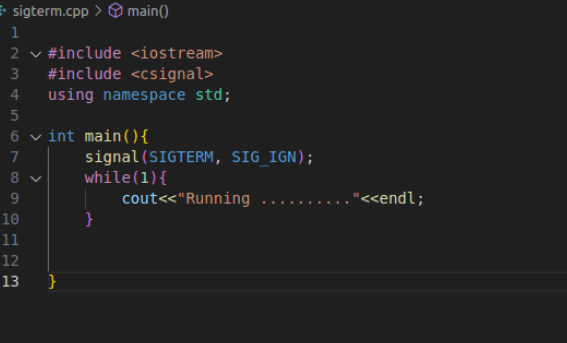
SIGINT

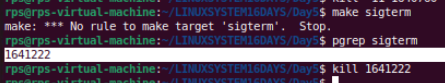


SIGSEGV

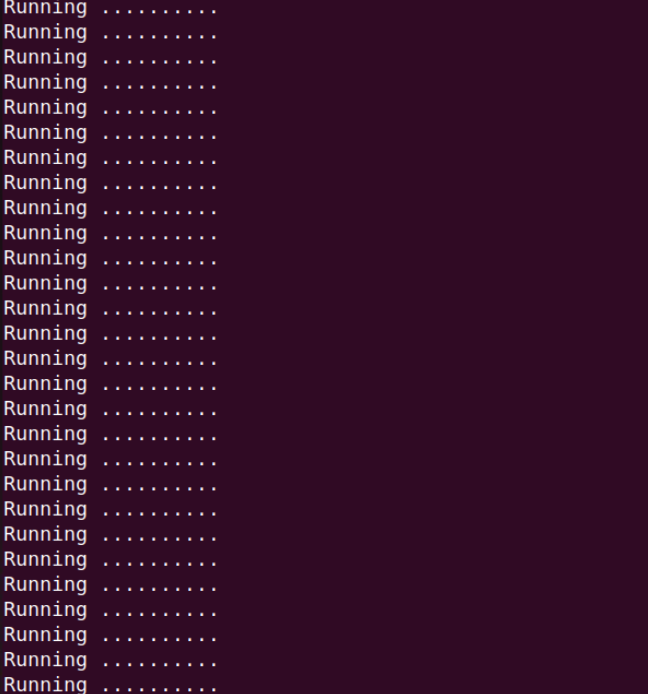


SIGTERM

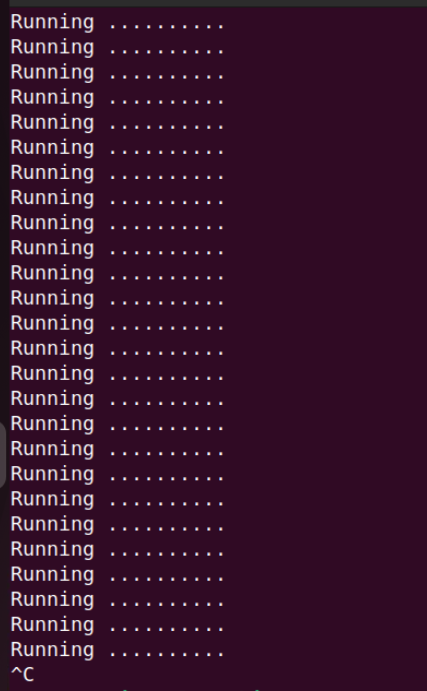




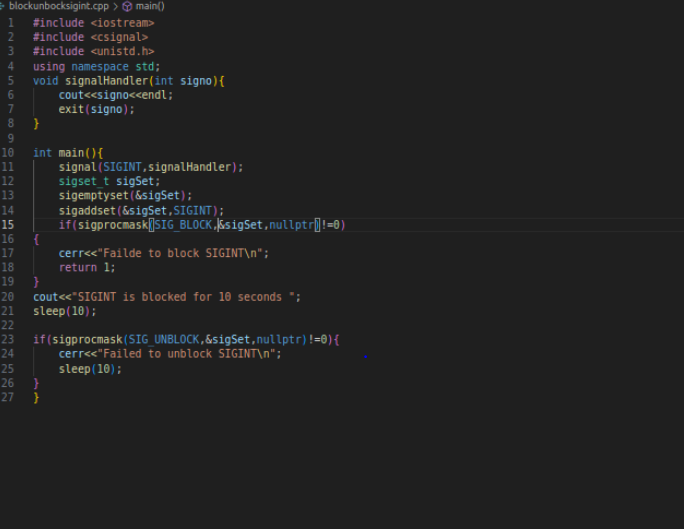
**After killing the process it will still run and ignore sigterm**

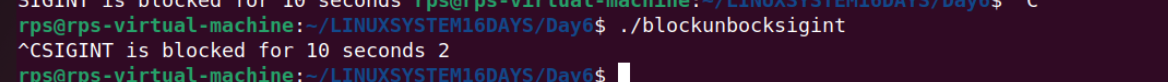


**To terminate program use ctrl +c**



SIGINT BLOCK AND UNBLOCK

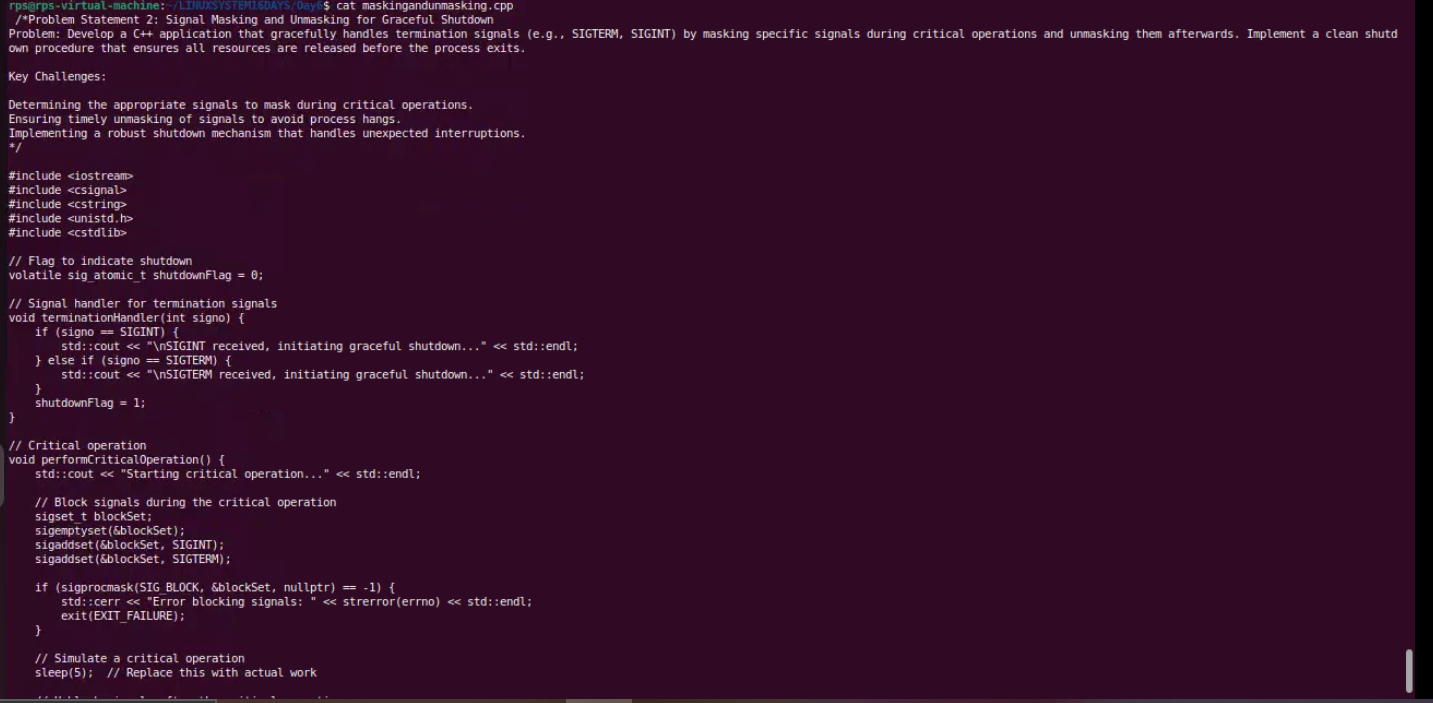


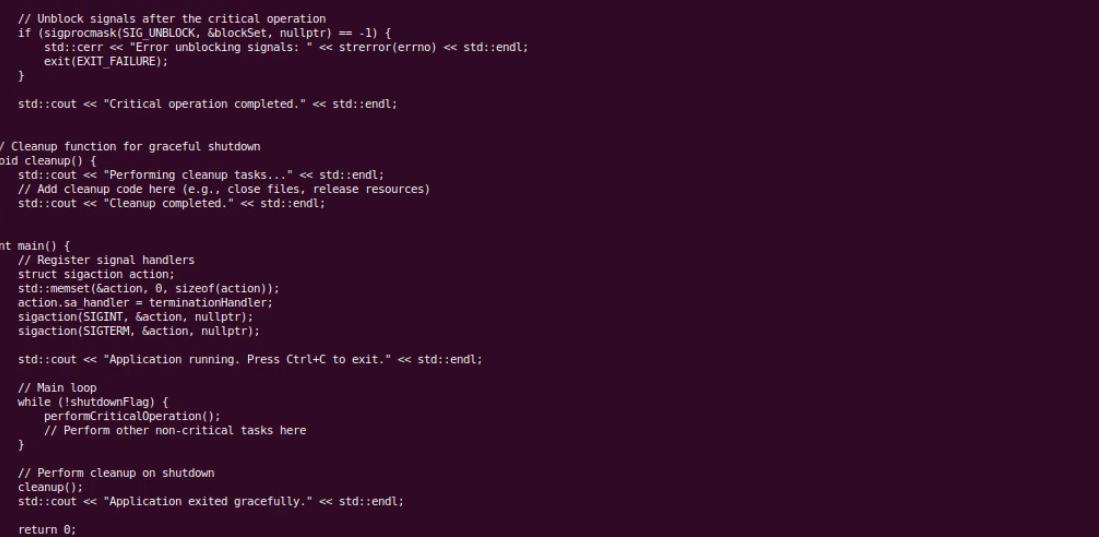


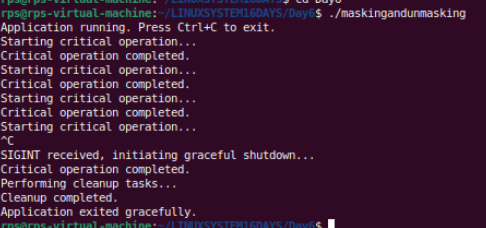
Problem Statement 2: Signal Masking and Unmasking for Graceful Shutdown  
Problem: Develop a C++ application that gracefully handles termination signals (e.g., SIGTERM, SIGINT) by masking specific signals during critical operations and unmasking them afterwards. Implement a clean shutdown procedure that ensures all resources are released before the process exits.

Key Challenges:

Determining the appropriate signals to mask during critical operations.  
Ensuring timely unmasking of signals to avoid process hangs.  
Implementing a robust shutdown mechanism that handles unexpected interruptions.







Problem Statement 3: Signal Masking and Unmasking for Error Handling  
Problem: Create a C++ application that uses signal masking and unmasking to handle errors gracefully. Mask specific signals during error handling routines to prevent recursive signal delivery. Implement a mechanism to log error details and perform necessary cleanup actions before re-enabling the masked signals.

Key Challenges:

Identifying the appropriate signals to mask during error handling.  
Preventing infinite recursion of signal handlers.  
Ensuring proper error logging and resource cleanup.



