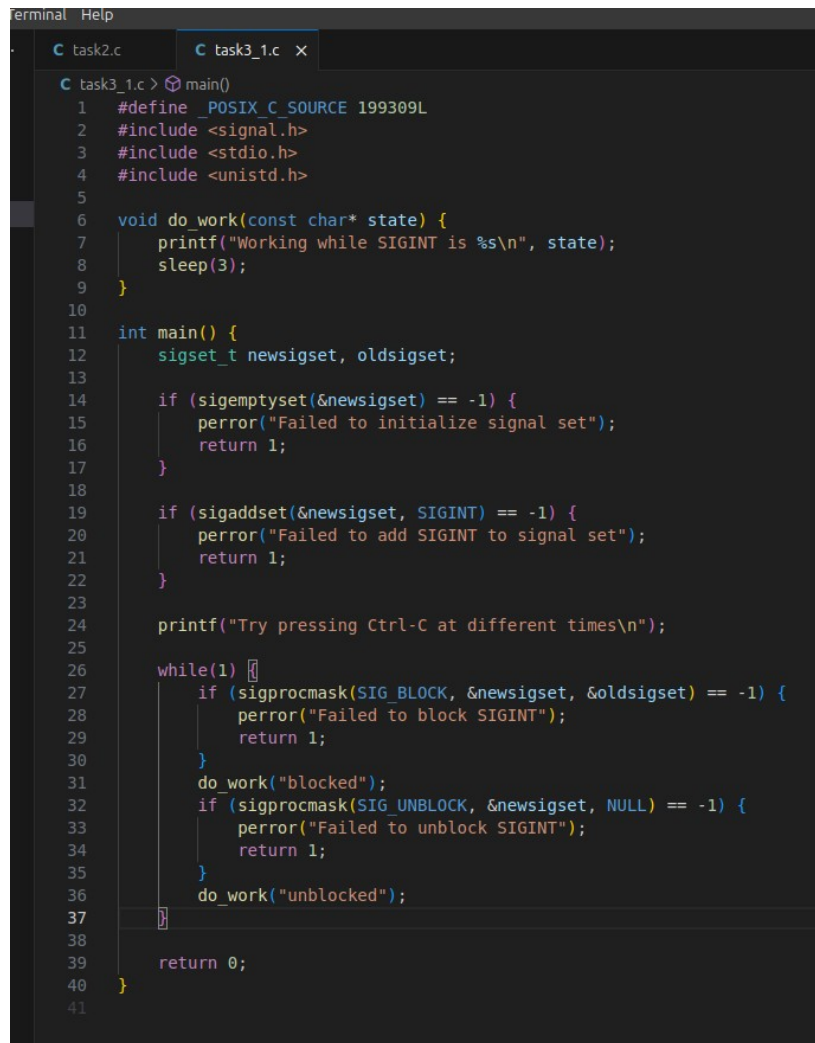


Task 2.1:

```
C task2.c > main()
1  #include <signal.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <unistd.h>
6  #include <sys/types.h>
7
8  int main() {
9      char name[100];
10     pid_t parent_pid = getppid();
11
12     printf("Enter your name: ");
13     fgets(name, sizeof(name), stdin);
14     name[strcspn(name, "\n")] = 0;
15
16     printf("Sending signal to parent process (PID: %d)\n", parent_pid);
17
18     if (kill(parent_pid, SIGUSR1) == -1) {
19         perror("Failed to send SIGUSR1 signal");
20         return 1;
21     }
22
23     printf("Hello %s, successfully sent SIGUSR1 signal to parent process %d\n",
24           name, parent_pid);
25     return 0;
26 }
27
```

Explanation: I first declare a main function that gets the parent process ID using `getppid()` function. After getting the parent process ID, I prompt the user to enter their name using `fgets()`, removes the trailing newline from the input using `strcspn()`, and then attempts to send a `SIGUSR1` signal to the parent process using the `kill()` function. Then i use error handling with if statements to check if the `kill()` function succeeded (returns 0) or failed (returns -1).

Task3.1 :



```
terminal Help
C task2.c C task3_1.c X
C task3_1.c > main()
1 #define _POSIX_C_SOURCE 199309L
2 #include <signal.h>
3 #include <stdio.h>
4 #include <unistd.h>
5
6 void do_work(const char* state) {
7     printf("Working while SIGINT is %s\n", state);
8     sleep(3);
9 }
10
11 int main() {
12     sigset_t newsigset, oldsigset;
13
14     if (sigemptyset(&newsigset) == -1) {
15         perror("Failed to initialize signal set");
16         return 1;
17     }
18
19     if (sigaddset(&newsigset, SIGINT) == -1) {
20         perror("Failed to add SIGINT to signal set");
21         return 1;
22     }
23
24     printf("Try pressing Ctrl-C at different times\n");
25
26     while(1) {
27         if (sigprocmask(SIG_BLOCK, &newsigset, &oldsigset) == -1) {
28             perror("Failed to block SIGINT");
29             return 1;
30         }
31         do_work("blocked");
32         if (sigprocmask(SIG_UNBLOCK, &newsigset, NULL) == -1) {
33             perror("Failed to unblock SIGINT");
34             return 1;
35         }
36         do_work("unblocked");
37     }
38
39     return 0;
40 }
41
```

explanation: This code demonstrates signal blocking and unblocking in a UNIX/Linux environment. The program creates a signal set and specifically adds SIGINT (the signal generated by Ctrl-C) to it. It then enters an infinite loop where it alternates between two states: one where SIGINT is blocked and another where it's unblocked. During each state, it performs some work (simulated by a 3-second sleep) and prints whether SIGINT is currently blocked or unblocked. When SIGINT is blocked and a user presses Ctrl-C, the signal is held until the program enters the unblocked state. Conversely, when SIGINT is unblocked and Ctrl-C is pressed, the program terminates immediately. This behavior is achieved using sigprocmask() to control signal blocking, while sigemptyset() and sigaddset() are used to manage the signal set

3.2) : Yes, it is possible for pipe1 to exist while pipe2 does not after a call to makepair due to the non-atomic nature of the operation. This situation can occur in the brief interval between the two mkfifo() calls, where the first call successfully creates pipe1, but the second call fails to create pipe2

1

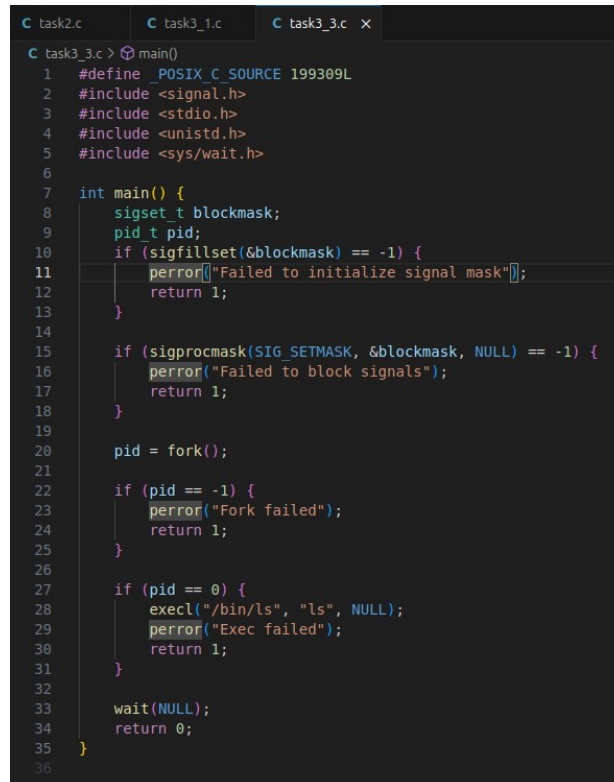
. Even though the code attempts to handle this by blocking signals during pipe creation and implementing cleanup procedures, system-level issues like resource limitations, permission changes, or interference from other processes could cause pipe2's creation to fail while leaving pipe1 intact. The cleanup code attempts to unlink both pipes if either creation fails, but if the unlink operation itself fails for pipe1 (due to sudden permission changes or system issues), pipe1 would remain while pipe2 was never created

3.3)

a) No, a makepair return value of 0 does not guarantee that FIFOs corresponding to pipe1 and pipe2 are available on return. This is because:

- Another process might delete or modify the FIFOs after makepair creates them but before the calling process uses them
- Permission changes could occur between creation and usage
- System resource limitations might affect access after creation

b)



```
C task2.c C task3_1.c C task3_3.c X
C task3_3.c > main()
1  #define _POSIX_C_SOURCE 199309L
2  #include <signal.h>
3  #include <stdio.h>
4  #include <unistd.h>
5  #include <sys/wait.h>
6
7  int main() {
8      sigset_t blockmask;
9      pid_t pid;
10     if (sigfillset(&blockmask) == -1) {
11         perror("Failed to initialize signal mask");
12         return 1;
13     }
14
15     if (sigprocmask(SIG_SETMASK, &blockmask, NULL) == -1) {
16         perror("Failed to block signals");
17         return 1;
18     }
19
20     pid = fork();
21
22     if (pid == -1) {
23         perror("Fork failed");
24         return 1;
25     }
26
27     if (pid == 0) {
28         execl("/bin/ls", "ls", NULL);
29         perror("Exec failed");
30         return 1;
31     }
32
33     wait(NULL);
34     return 0;
35 }
36
```

explanation: I wrote a program that first blocks all signals using sigfillset() to create a signal mask (blockmask) and then applies this mask using sigprocmask() to prevent any signals from interrupting the parent process. After blocking signals, I used fork() to create a child process. In the child process (when pid == 0), I executed the ls command using execl("/bin/ls", "ls", NULL) which displays the contents of the current directory. Meanwhile, the parent process waits for the child to complete using wait(NULL). If any step fails (like signal blocking, forking, or executing ls), the program reports the error using perror() and returns with an error code. The program demonstrates how to protect a parent process from signals while allowing a child process to execute a command normally.

4.1) Example 12 didn't use `fprintf` or `strlen` in the signal handler because signal handlers should only use async-signal-safe functions. Functions like `fprintf` and `strlen` are not async-signal-safe because:

1. `fprintf` uses buffered I/O and maintains internal data structures that could be in an inconsistent state when the signal handler interrupts the main program
1
2. These functions might internally use `malloc()` or other non-reentrant functions that could cause deadlocks or data corruption if interrupted

Instead, the example uses `write()` which is async-signal-safe and guaranteed to be atomic up to a certain size. The message length is pre-calculated using `sizeof` rather than `strlen()` to avoid any unsafe function calls during signal handling