ÇAĞRI YILDIZ 1901042630

CSE 344

HOMEWORK 1 REPORT

This report includes the following components:

1. ***Homework Requirements:*** A detailed overview of the assignment's expectations and objectives.
2. ***Program Execution and Output Description:*** An explanation of how the program operates, including descriptions of the outputs observed during its execution.

Homework Requirements:

1. Program Setup and Input Validation

```c
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <size_of_array>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    char *endptr; // Pointer to the end of the parsed string
    long size = strtol(argv[1], &endptr, 10); // Convert string to long

    // Check for non-numeric input, no input, and integer-specific errors
    if (endptr == argv[1] || *endptr != '\0' || errno == ERANGE) {
        fprintf(stderr, "Invalid number: %s\n", argv[1]);
        exit(EXIT_FAILURE);
    }

    // Check for non-positive size of the array
    if (size <= 0) {
        fprintf(stderr, "Array size must be a positive integer, given: %ld\n", size);
        exit(EXIT_FAILURE);
    }
```

Requirement Covered: "Create a program that takes an integer argument."
Functionality: This section checks if the ***command line*** input argument (array size) is valid, handling non-integer and non-positive inputs by providing clear error messages and terminating if the input is not valid.

2. Array Initialization and Random Number Generation

```c

    int numbers[size];
    for (int i = 0; i < size; i++) {
        numbers[i] = rand() % 10;  // Initialize array with values
    }

    printf("Array : [");
    for (int i = 0; i < size-1; i++) {
        printf("%d, ",numbers[i]);  // Printing array
    }
    printf("%d]\n\n",numbers[size-1]);
```

Requirement Covered: Part of "Send an array of random numbers to the first FIFO."
Functionality: Initializes an array with random values and prints it. This ensures that the values which will be sent to the FIFOs are visible for debugging and verification.

3. Signal Handler Setup

```
110
111     setup_signal_handlers();
112
```

```
50  // Setup signal handlers for SIGCHLD and SIGINT
51  void setup_signal_handlers() {
52      struct sigaction sa; // Struct to specify the action to be associated with SIGCHLD and SIGINT
53      sigemptyset(&sa.sa_mask); // Initialize the signal set to empty, ensuring no signals are blocked during the execution of the handler
54      sa.sa_flags = SA_RESTART; // Use SA_RESTART to make certain system calls restartable across signals
55
56      // Set sigchld_handler as the handler function for SIGCHLD
57      sa.sa_handler = sigchld_handler;
58      // Apply the signal action to SIGCHLD
59      if (sigaction(SIGCHLD, &sa, NULL) == -1) {
60          perror("Error setting SIGCHLD handler"); // If sigaction fails, print error message
61          exit(EXIT_FAILURE); // Exit if there is an error setting the signal
62      }
63
64      // Set signal_handler as the handler function for SIGINT
65      sa.sa_handler = signal_handler;
66      // Apply the signal action to SIGINT
67      if (sigaction(SIGINT, &sa, NULL) == -1) {
68          perror("Error setting SIGINT handler"); // If sigaction fails, print error message
69          exit(EXIT_FAILURE); // Exit if there is an error setting the signal
70      }
71
72      // Register cleanup_resources to be called upon program exit
73      atexit(cleanup_resources); // This ensures that cleanup_resources is called when exit() is called anywhere in the program
74  }
```

Requirement Covered: "Set a signal handler for SIGCHLD in the parent process to handle child process termination."
Functionality: Configures signal handlers for both SIGCHLD and SIGINT. It ensures that child processes are reaped properly and resources are cleaned up if the program is interrupted.

4. FIFO Creation

```
113     // Create FIFOs for IPC
114     if (mkfifo(FIFO1, 0666) == -1) {
115         perror("Failed to create FIFO1");
116         exit(EXIT_FAILURE);
117     }
118
119     if (mkfifo(FIFO2, 0666) == -1) {
120         perror("Failed to create FIFO2");
121         exit(EXIT_FAILURE);
122     }
```

Requirement Covered: "Create two FIFOs (named pipes)."
Functionality: Attempts to create two FIFOs for inter-process communication. If either FIFO cannot be created, it outputs an error message and exits.

5. Fork and Process Implementation

```
124     // Forking to create Child 1
125     pid_t pid1 = fork();
126     if (pid1 == -1) {
127         perror("Failed to fork child 1");
128         exit(EXIT_FAILURE);
129     }
130
131     if (pid1 == 0) { // Child 1 process
132         sleep(10); // Delay to simulate processing time
133         int fifo1 = open(FIFO1, O_RDONLY);
134         int sum = 0, temp;
135
136         // Read integers from FIFO1 and calculate their sum
```

```
151     // Forking to create Child 2
152     pid_t pid2 = fork();
153     if (pid2 == -1) {
154         perror("Failed to fork child 2");
155         exit(EXIT_FAILURE);
156     }
157
158     if (pid2 == 0) { // Child 2 process
159         sleep(10);
160         int fifo2 = open(FIFO2, O_RDONLY);
161         int sum;
162         char command[10];
163
164         // Read the sum and command from FIFO2
```

Requirement Covered: "Use the fork() system call to create two child processes and assign each to a FIFO."
Functionality: Creates two child processes. Each child is assigned a specific task with FIFOs—Child 1 reads and sums the numbers; Child 2 reads the sum and a command, then performs a multiplication.

6. Loop for Monitoring Child Process Termination

```
191     // Wait for both children to terminate
192     while (children_terminated < 2) {
193         printf("Parent process proceeding\n");
194         sleep(2);
195     }
```

Requirement Covered: This loop fulfills the assignment's requirements to "Enter a loop, printing a message containing 'proceeding' every two seconds." It uses a simple while loop to check if the number of terminated children has reached the total number of children spawned (2 in this case).
Functionality: The loop continuously checks the children_terminated counter, which is incremented each time a child process is reaped. During each iteration of the loop, it prints "Parent process proceeding" and then pauses for two seconds using sleep(2), ensuring the message is printed every two seconds as required.

7. Signal Handler with waitpid()

```
36  // SIGCHLD handler to catch terminated children, print their status, and count them
37  void sigchld_handler(int sig) {
38      int status; // Variable to store the exit status of the terminated child
39      pid_t pid;  // Variable to store the PID of the terminated child
40
41      // Loop to reap all terminated children without blocking
42      while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
43          // Print the PID and exit status of the child process
44          printf("Child with PID %d exited with status %d\n", pid, WEXITSTATUS(status));
45          // Increment the count of terminated children
46          children_terminated++;
47      }
48  }
```

Requirement Covered: "The signal handler should call waitpid() to reap the terminated child process, print out the process ID of the exited child, and increment a counter."
Functionality: This signal handler is triggered by SIGCHLD, indicating that a child process has terminated. The handler uses waitpid() with the WNOHANG option, which allows it to reap any child processes that have terminated without blocking if there are no terminated children at that moment. This function also prints the PID and exit status of each reaped child and increments the global counter children_terminated.

8. Error Handling and Program Completion

```
191     // Wait for both children to terminate
192     while (children_terminated < 2) {
193         printf("Parent process proceeding\n");
194         sleep(2);
195     }
196
197     printf("All child processes have terminated.\n");
198
199     return EXIT_SUCCESS;
200 }
```

Requirement Covered: “When the counter reaches the number of children originally spawned, the program exits.”
Functionality: Monitors the termination of child processes and keeps the user informed with a "proceeding" message every two seconds. It finalizes with a message once all children have terminated, ensuring all processes have completed as expected.

Child Process 1: Reading from FIFO1 and Writing to FIFO2

```
130
131     if (pid1 == 0) { // Child 1 process
132         sleep(10); // Delay to simulate processing time
133         int fifo1 = open(FIFO1, O_RDONLY);
134         int sum = 0, temp;
135
136         // Read integers from FIFO1 and calculate their sum
137         while (read(fifo1, &temp, sizeof(temp)) > 0) {
138             sum += temp;
139         }
140         printf("summation result: %d\n", sum);
141
142         // Send sum to FIFO2 and a command to multiply
143         int fifo2 = open(FIFO2, O_WRONLY);
144         write(fifo2, &sum, sizeof(sum));
145         write(fifo2, "multiply", sizeof("multiply"));
146         close(fifo1);
147         close(fifo2);
148         exit(EXIT_SUCCESS);
149     }
```

Requirement Covered: Child Process 1 responsibilities.
Functionality: Opens the first FIFO, reads integers, calculates their sum, then writes this sum and a command to the second FIFO for processing by Child Process 2. Implements basic error handling for opening FIFOs and writing data.

Child Process 2: Reading from FIFO2 and Performing Multiplication

```
151     // Forking to create Child 2
152     pid_t pid2 = fork();
153     if (pid2 == -1) {
154         perror("Failed to fork child 2");
155         exit(EXIT_FAILURE);
156     }
157
158     if (pid2 == 0) { // Child 2 process
159         sleep(10);
160         int fifo2 = open(FIFO2, O_RDONLY);
161         int sum;
162         char command[10];
163
164         // Read the sum and command from FIFO2
165         read(fifo2, &sum, sizeof(sum));
166         read(fifo2, command, sizeof(command));
167
168         // Check command and perform multiplication if correct
169         if (strcmp(command, "multiply") == 0) {
170             int product = 1;
171             int finalSum = 0;
172             for (int i = 0; i < size; i++) {
173                 product *= numbers[i];
174             }
175             finalSum = sum + product;
176             printf("multiplication result: %d\n", product);
177             printf("FINAL RESULT => %d\n",finalSum);
178         }
179
```

Requirement Covered: Child Process 2 responsibilities.
Functionality: Opens the second FIFO, reads the sum and a command, performs multiplication if the command is "multiply". Provides an example of handling commands sent through IPC.

Bonus Section

```c
// SIGCHLD handler to catch terminated children, print their status, and count them
void sigchld_handler(int sig) {
    int status; // Variable to store the exit status of the terminated child
    pid_t pid;  // Variable to store the PID of the terminated child

    // Loop to reap all terminated children without blocking
    while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
        // Print the PID and exit status of the child process
        printf("Child with PID %d exited with status %d\n", pid, WEXITSTATUS(status));
        // Increment the count of terminated children
        children_terminated++;
    }
}
```

Zombie Protection Method: This is implemented using the sigchld_handler function which calls waitpid() with WNOHANG to reap any zombie processes created as child processes exit, without blocking the parent process.

Printing Exit Statuses of All Processes: The exit statuses are printed within the sigchld_handler:

Error Scenarios

1. Errors in Creating FIFOs or Data/Command Transmission:

```c
    // Create FIFOs for IPC
    if (mkfifo(FIFO1, 0666) == -1) {
        perror("Failed to create FIFO1");
        exit(EXIT_FAILURE);
    }

    if (mkfifo(FIFO2, 0666) == -1) {
        perror("Failed to create FIFO2");
        exit(EXIT_FAILURE);
    }
```

These lines check if there is an error in creating either of the FIFOs. If an error occurs, it uses perror() to print an appropriate error message that includes the reason for the failure and then exits the program. This prevents the program from proceeding with uninitialized FIFOs.

```c
    // Parent process writing to FIFO1
    int fifo1 = open(FIFO1, O_WRONLY);
    for (int i = 0; i < size; i++) {
        if (write(fifo1, &numbers[i], sizeof(numbers[i])) == -1) {
            perror("Failed to write to FIFO1");
            exit(EXIT_FAILURE);
        }
    }
    close(fifo1);
```

By checking the return value of each write() call, you can ensure that errors in data transmission are caught and handled promptly, preventing the child process from proceeding with incomplete data transmission.

2. Errors in Child Process Completion:

```
177         close(fifo2);
178         exit(EXIT_SUCCESS);
179     }
```

Child processes exit with EXIT_SUCCESS after completing their tasks.

3. Management of Counter Value and Exit Statuses:

```
while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
    // Print the PID and exit status of the child process
    printf("Child with PID %d exited with status %d\n", pid, WEXITSTATUS(status));
    // Increment the count of terminated children
    children_terminated++;
}
```

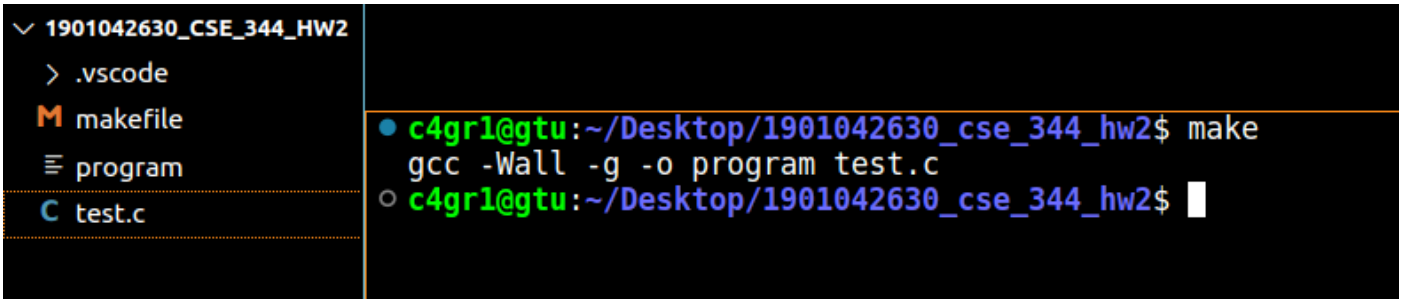
```
191     // Wait for both children to terminate
192     while (children_terminated < 2) {
193         printf("Parent process proceeding\n");
194         sleep(2);
195     }
196
```

The program increments a counter for each child process that terminates and checks its exit status.
This loop in sigchld_handler reaps child processes without blocking, using WNOHANG.
Proper management of the counter ensures that the parent knows when all child processes have finished

Program Execution and Output Description

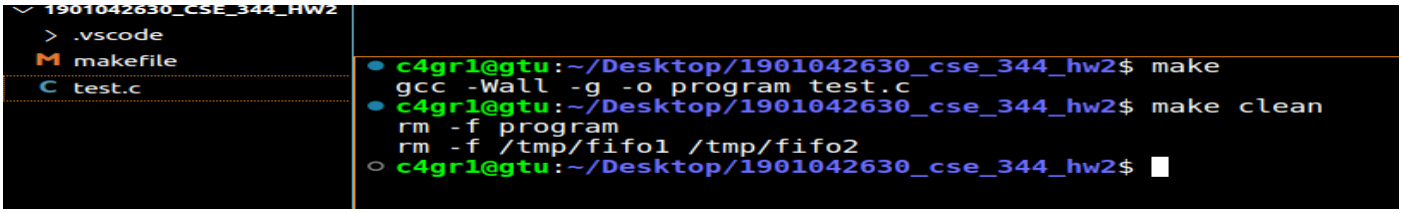
1. make or make all

This command compiles your source file (test.c) into the executable named program. It uses the GCC compiler with debugging symbols and all warnings enabled, which is useful for development and debugging.



2. make clean

This command removes the compiled executable and any FIFO files created during runtime. It's helpful for cleaning up your project directory and ensuring that subsequent builds start from a clean state.



3. make run

This command runs the compiled executable. You can provide command line arguments to your program using the ARGS variable. For instance, if your program accepts an array size as an argument, you can pass it like so:

```
c4gr1@gtu:~/Desktop/1901042630_cse_344_hw2$ make run ARGS=5
./program 5
Array : [1, 4, 8, 0, 2]

Parent process proceeding
summation result: 15
Resources cleaned up.
Child with PID 11242 exited with status 0
Parent process proceeding
multiplication result: 0
FINAL RESULT => 15
Resources cleaned up.
Child with PID 11243 exited with status 0
All child processes have terminated.
Resources cleaned up.
c4gr1@gtu:~/Desktop/1901042630_cse_344_hw2$ make run ARGS=5
./program 5
Array : [5, 8, 2, 3, 6]

Parent process proceeding
summation result: 24
Resources cleaned up.
Child with PID 11322 exited with status 0
Parent process proceeding
multiplication result: 1440
FINAL RESULT => 1464
Resources cleaned up.
Child with PID 11323 exited with status 0
All child processes have terminated.
Resources cleaned up.
c4gr1@gtu:~/Desktop/1901042630_cse_344_hw2$
```

```
c4gr1@gtu:~/Desktop/1901042630_cse_344_hw2$ make run ARGS=10
./program 10
Array : [3, 1, 4, 1, 5, 8, 7, 1, 9, 9]

Parent process proceeding
summation result: 48
Resources cleaned up.
Child with PID 11402 exited with status 0
Parent process proceeding
multiplication result: 272160
FINAL RESULT => 272208
Resources cleaned up.
Child with PID 11403 exited with status 0
All child processes have terminated.
Resources cleaned up.
c4gr1@gtu:~/Desktop/1901042630_cse_344_hw2$ make run ARGS=10
./program 10
Array : [2, 1, 8, 1, 5, 5, 9, 7, 6, 3]

Parent process proceeding
summation result: 47
Resources cleaned up.
Child with PID 11480 exited with status 0
Parent process proceeding
multiplication result: 453600
FINAL RESULT => 453647
Resources cleaned up.
Child with PID 11481 exited with status 0
All child processes have terminated.
Resources cleaned up.
c4gr1@gtu:~/Desktop/1901042630_cse_344_hw2$
```

4. make valgrind

This command runs your program under Valgrind to check for memory leaks. This is crucial for ensuring your program manages memory correctly and doesn't have leaks, which can cause problems in production environments.

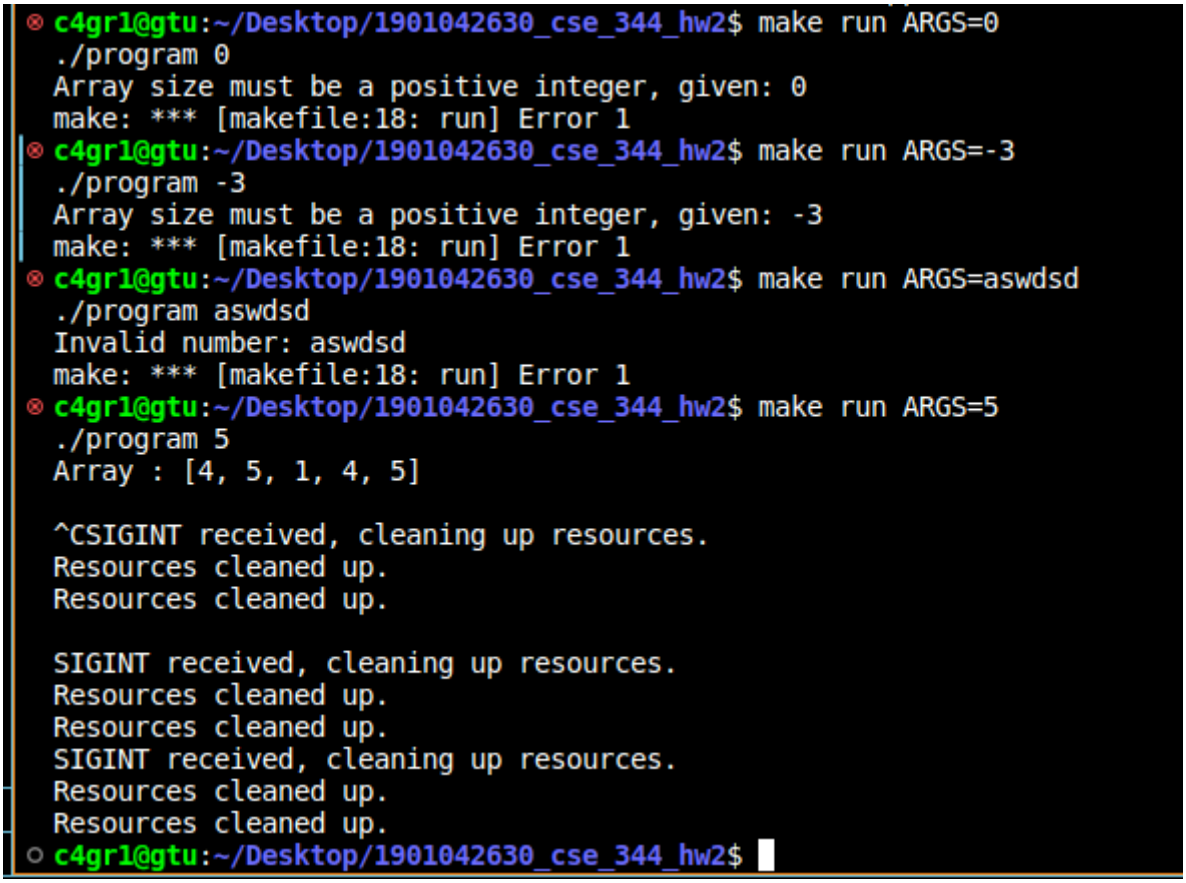
```
c4gr1@gtu:~/Desktop/1901042630_cse_344_hw2$ make valgrind ARGS=7
valgrind --leak-check=full --show-leak-kinds=all ./program 7
==11570== Memcheck, a memory error detector
==11570== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==11570== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==11570== Command: ./program 7
==11570==
Array : [0, 4, 8, 3, 2, 7, 4]

Parent process proceeding
summation result: 28
multiplication result: 0
FINAL RESULT => 28
Resources cleaned up.
Resources cleaned up.
==11586==
==11585==
==11585== HEAP SUMMARY:
==11585==     in use at exit: 0 bytes in 0 blocks
==11585==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==11585==
==11585== All heap blocks were freed -- no leaks are possible
==11585==
==11585== For lists of detected and suppressed errors, rerun with: -s
==11585== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==11586== HEAP SUMMARY:
==11586==     in use at exit: 0 bytes in 0 blocks
==11586==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==11586==
==11586== All heap blocks were freed -- no leaks are possible
==11586==
==11586== For lists of detected and suppressed errors, rerun with: -s
==11586== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
Child with PID 11586 exited with status 0
Parent process proceeding
Child with PID 11585 exited with status 0
All child processes have terminated.
Resources cleaned up.
==11570==
==11570== HEAP SUMMARY:
==11570==     in use at exit: 0 bytes in 0 blocks
==11570==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==11570==
==11570== All heap blocks were freed -- no leaks are possible
==11570==
==11570== For lists of detected and suppressed errors, rerun with: -s
==11570== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
c4gr1@gtu:~/Desktop/1901042630_cse_344_hw2$ make valgrind ARGS=1
```

```
c4gr1@gtu:~/Desktop/1901042630_cse_344_hw2$ make valgrind ARGS=1
valgrind --leak-check=full --show-leak-kinds=all ./program 1
==11642== Memcheck, a memory error detector
==11642== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==11642== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==11642== Command: ./program 1
==11642==
Array : [9]

Parent process proceeding
summation result: 9
multiplication result: 9
FINAL RESULT => 18
Resources cleaned up.
Resources cleaned up.
==11658==
==11657==
==11657== HEAP SUMMARY:
==11657==     in use at exit: 0 bytes in 0 blocks
==11657==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==11657==
==11657== All heap blocks were freed -- no leaks are possible
==11657==
==11657== For lists of detected and suppressed errors, rerun with: -s
==11657== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==11658== HEAP SUMMARY:
==11658==     in use at exit: 0 bytes in 0 blocks
==11658==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==11658==
==11658== All heap blocks were freed -- no leaks are possible
==11658==
==11658== For lists of detected and suppressed errors, rerun with: -s
==11658== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
Child with PID 11658 exited with status 0
Parent process proceeding
Parent process proceeding
Child with PID 11657 exited with status 0
All child processes have terminated.
Resources cleaned up.
==11642==
==11642== HEAP SUMMARY:
==11642==     in use at exit: 0 bytes in 0 blocks
==11642==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==11642==
==11642== All heap blocks were freed -- no leaks are possible
==11642==
==11642== For lists of detected and suppressed errors, rerun with: -s
==11642== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
c4gr1@gtu:~/Desktop/1901042630_cse_344_hw2$
```

Invalid Inputs & CTRL+C Handling



THANKS FOR READING

ÇAĞRI YILDIZ

1901042630