CSE – 344 Homework 2 Report

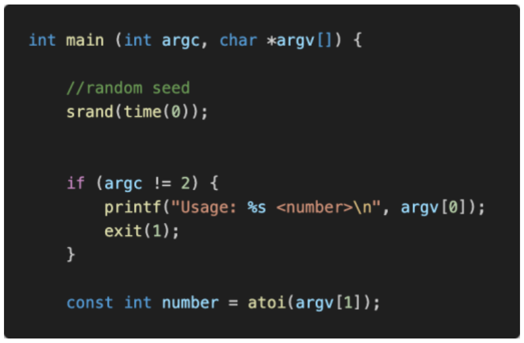
Süleyman Burak Yaşar
1901042662

Usage Of Makefile

make: Compiles the code
run: Run the program -> make run “number”
run2: Run the program with valgrind -> make run2 “number”
clean: Remove test program

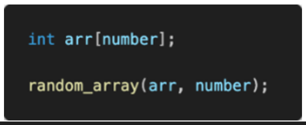
Taking Number from User

From the terminal, a single number is taken as a parameter from the user. First, the number of entered parameters is checked to determine if only one parameter has been entered. Then, the entered parameter is converted to a number.



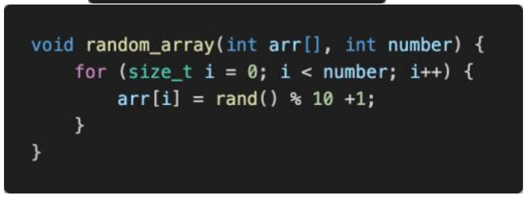
```
int main (int argc, char *argv[]) {

    //random seed
    srand(time(0));


    if (argc != 2) {
        printf("Usage: %s <number>\n", argv[0]);
        exit(1);
    }

    const int number = atoi(argv[1]);
```

Generate Random Array



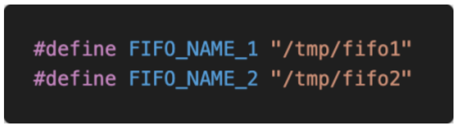
```
int arr[number];

random_array(arr, number);
```

An array consisting of random elements of a length equal to the number taken from the user is created. To prevent overflow during future multiplication, random numbers are chosen between 0 and 9.

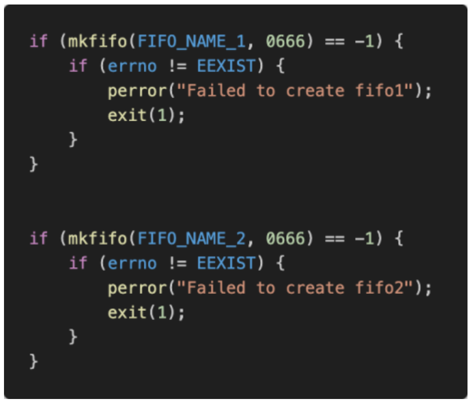


```
void random_array(int arr[], int number) {
    for (size_t i = 0; i < number; i++) {
        arr[i] = rand() % 10 +1;
    }
}
```

Creating Fifo



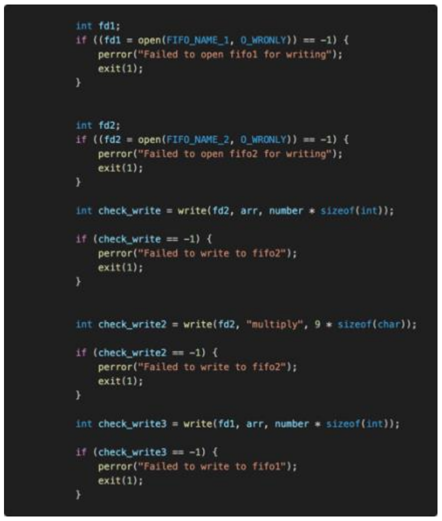
```c
#define FIFO_NAME_1 "/tmp/fifo1"
#define FIFO_NAME_2 "/tmp/fifo2"
```



```c
if (mkfifo(FIFO_NAME_1, 0666) == -1) {
    if (errno != EEXIST) {
        perror("Failed to create fifo1");
        exit(1);
    }
}


if (mkfifo(FIFO_NAME_2, 0666) == -1) {
    if (errno != EEXIST) {
        perror("Failed to create fifo2");
        exit(1);
    }
}
```

Fifo_1 and Fifo_2 are created using the mkfifo() function. If there is any error during creation, the program prints the error and terminates.

Write Array to Fifos

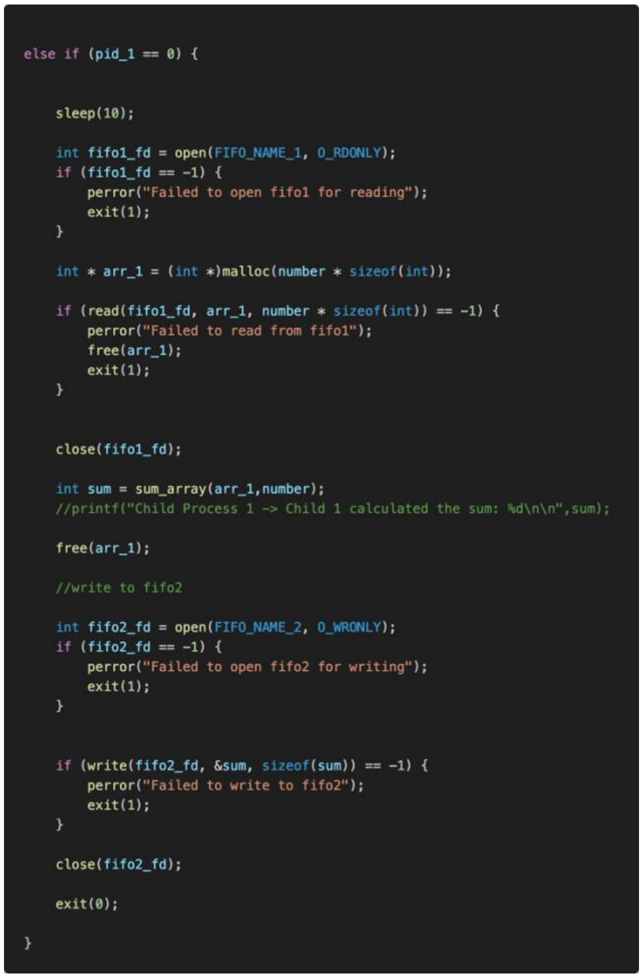


```c
int fd1;
if ((fd1 = open(FIFO_NAME_1, O_WRONLY)) == -1) {
    perror("Failed to open fifo1 for writing");
    exit(1);
}


int fd2;
if ((fd2 = open(FIFO_NAME_2, O_WRONLY)) == -1) {
    perror("Failed to open fifo2 for writing");
    exit(1);
}

int check_write = write(fd2, arr, number * sizeof(int));

if (check_write == -1) {
    perror("Failed to write to fifo2");
    exit(1);
}


int check_write2 = write(fd2, "multiply", 9 * sizeof(char));

if (check_write2 == -1) {
    perror("Failed to write to fifo2");
    exit(1);
}

int check_write3 = write(fd1, arr, number * sizeof(int));

if (check_write3 == -1) {
    perror("Failed to write to fifo1");
    exit(1);
}
```

Fifo_1 and fifo_2 are opened for writing.
The array created is written to Fifo_1 and Fifo_2.
The multiply command is written to Fifo_2.

Child 1

At the start of the child process, it is first put to sleep for 10 seconds using the sleep() function. Fifo_1 is opened for reading. Fifo_1 is read using the read() function, and the data is saved to a new array. The sum of the elements of this array is calculated using the sum_array() function and assigned to the sum variable. Fifo_2 is opened to write the calculated total. The total value is written to fifo_2. After this operation, the child process 1 completes its task and terminates.

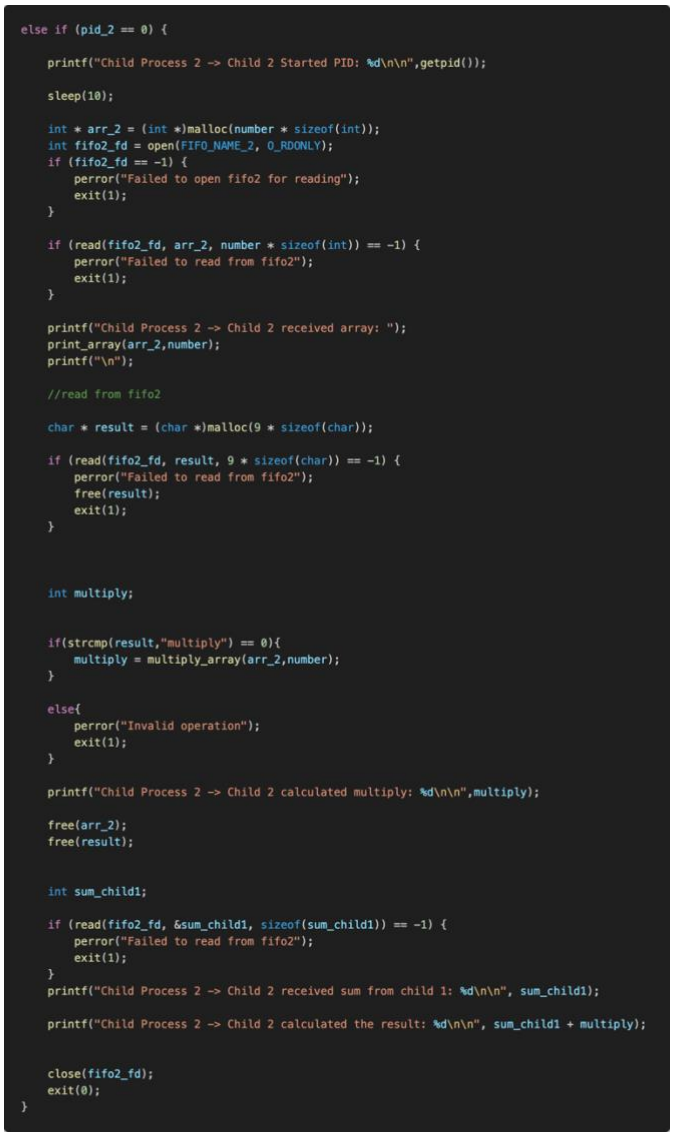


```c
else if (pid_1 == 0) {


    sleep(10);

    int fifo1_fd = open(FIFO_NAME_1, O_RDONLY);
    if (fifo1_fd == -1) {
        perror("Failed to open fifo1 for reading");
        exit(1);
    }

    int * arr_1 = (int *)malloc(number * sizeof(int));

    if (read(fifo1_fd, arr_1, number * sizeof(int)) == -1) {
        perror("Failed to read from fifo1");
        free(arr_1);
        exit(1);
    }


    close(fifo1_fd);

    int sum = sum_array(arr_1,number);
    //printf("Child Process 1 -> Child 1 calculated the sum: %d\n\n",sum);

    free(arr_1);

    //write to fifo2

    int fifo2_fd = open(FIFO_NAME_2, O_WRONLY);
    if (fifo2_fd == -1) {
        perror("Failed to open fifo2 for writing");
        exit(1);
    }


    if (write(fifo2_fd, &sum, sizeof(sum)) == -1) {
        perror("Failed to write to fifo2");
        exit(1);
    }

    close(fifo2_fd);

    exit(0);

}
```

Child 2

At the start of the child process, it is first put to sleep for 10 seconds using the sleep() function. Fifo_2 is opened for reading. First, the array sent by the parent is read using the read() function from Fifo_1 and saved to a new array. The second read operation retrieves the command sent by the parent and stores it in a variable. It checks if the read command is "multiply." If the command is correct, the multiplication of the elements of the array is calculated and stored in a variable.

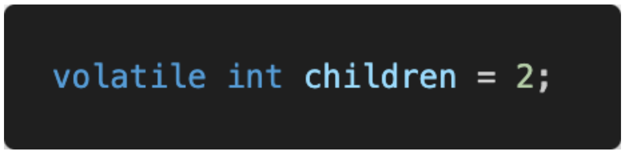
To read the total value sent by child_1, fifo_2 is read again. The read value is stored in a variable. At the end of the process, the total and multiplication values are printed, and the process terminates.



```c
else if (pid_2 == 0) {

    printf("Child Process 2 -> Child 2 Started PID: %d\n\n",getpid());

    sleep(10);

    int * arr_2 = (int *)malloc(number * sizeof(int));
    int fifo2_fd = open(FIFO_NAME_2, O_RDONLY);
    if (fifo2_fd == -1) {
        perror("Failed to open fifo2 for reading");
        exit(1);
    }

    if (read(fifo2_fd, arr_2, number * sizeof(int)) == -1) {
        perror("Failed to read from fifo2");
        exit(1);
    }

    printf("Child Process 2 -> Child 2 received array: ");
    print_array(arr_2,number);
    printf("\n");

    //read from fifo2

    char * result = (char *)malloc(9 * sizeof(char));

    if (read(fifo2_fd, result, 9 * sizeof(char)) == -1) {
        perror("Failed to read from fifo2");
        free(result);
        exit(1);
    }



    int multiply;


    if(strcmp(result,"multiply") == 0){
        multiply = multiply_array(arr_2,number);
    }

    else{
        perror("Invalid operation");
        exit(1);
    }

    printf("Child Process 2 -> Child 2 calculated multiply: %d\n\n",multiply);

    free(arr_2);
    free(result);


    int sum_child1;

    if (read(fifo2_fd, &sum_child1, sizeof(sum_child1)) == -1) {
        perror("Failed to read from fifo2");
        exit(1);
    }
    printf("Child Process 2 -> Child 2 received sum from child 1: %d\n\n", sum_child1);

    printf("Child Process 2 -> Child 2 calculated the result: %d\n\n", sum_child1 + multiply);


    close(fifo2_fd);
    exit(0);
}
```

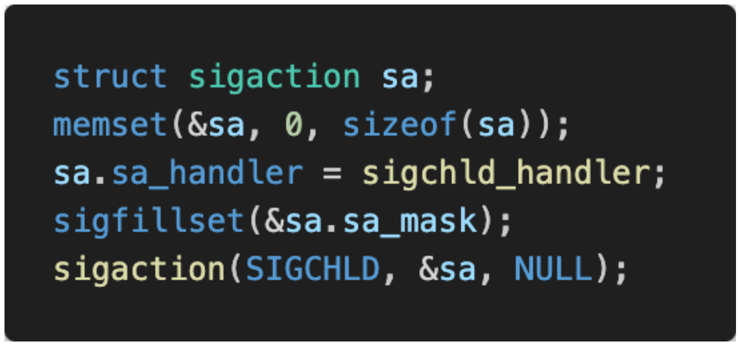
Signal Handling

Initially, the total number of children to be created is defined.

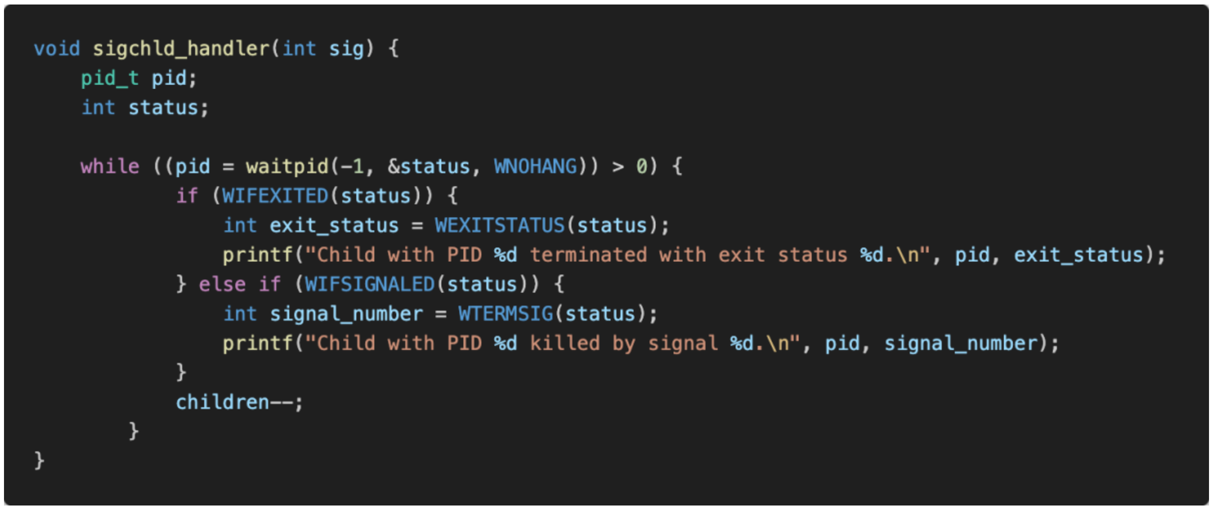


```c
volatile int children = 2;
```

Inside the main function, the necessary arrangements were made to catch signals coming from the child process.

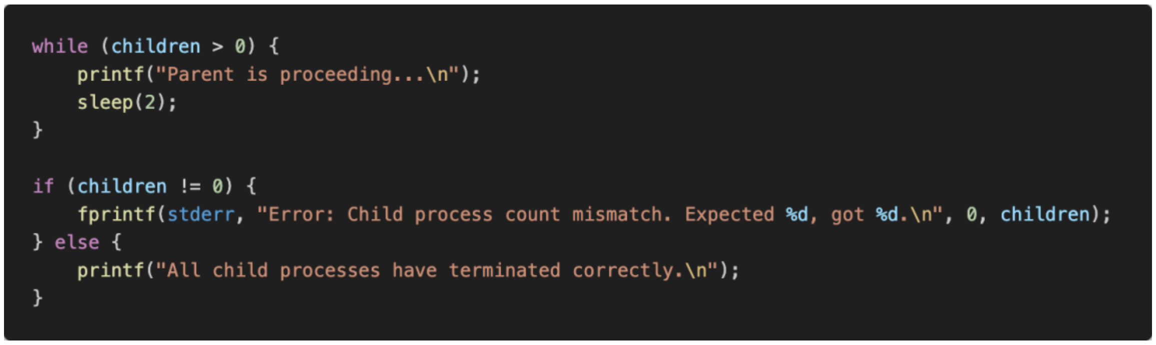


```c
struct sigaction sa;
memset(&sa, 0, sizeof(sa));
sa.sa_handler = sigchld_handler;
sigfillset(&sa.sa_mask);
sigaction(SIGCHLD, &sa, NULL);
```



```c
void sigchld_handler(int sig) {
    pid_t pid;
    int status;

    while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
            if (WIFEXITED(status)) {
                int exit_status = WEXITSTATUS(status);
                printf("Child with PID %d terminated with exit status %d.\n", pid, exit_status);
            } else if (WIFSIGNALED(status)) {
                int signal_number = WTERMSIG(status);
                printf("Child with PID %d killed by signal %d.\n", pid, signal_number);
            }
            children--;
        }
}
```

The sigchld_handler() function is a signal handler that manages the status of child processes when a SIGCHLD signal is received by the parent process. This handler uses the `waitpid` function to non-blockingly query all child processes whose statuses have not yet been reported (`WNOHANG` option). If a child process has exited normally, it prints the exit status; if terminated by a signal, it prints which signal caused the termination. It also decreases the child process counter, updating the number of active processes. It is also provide zombie protection.

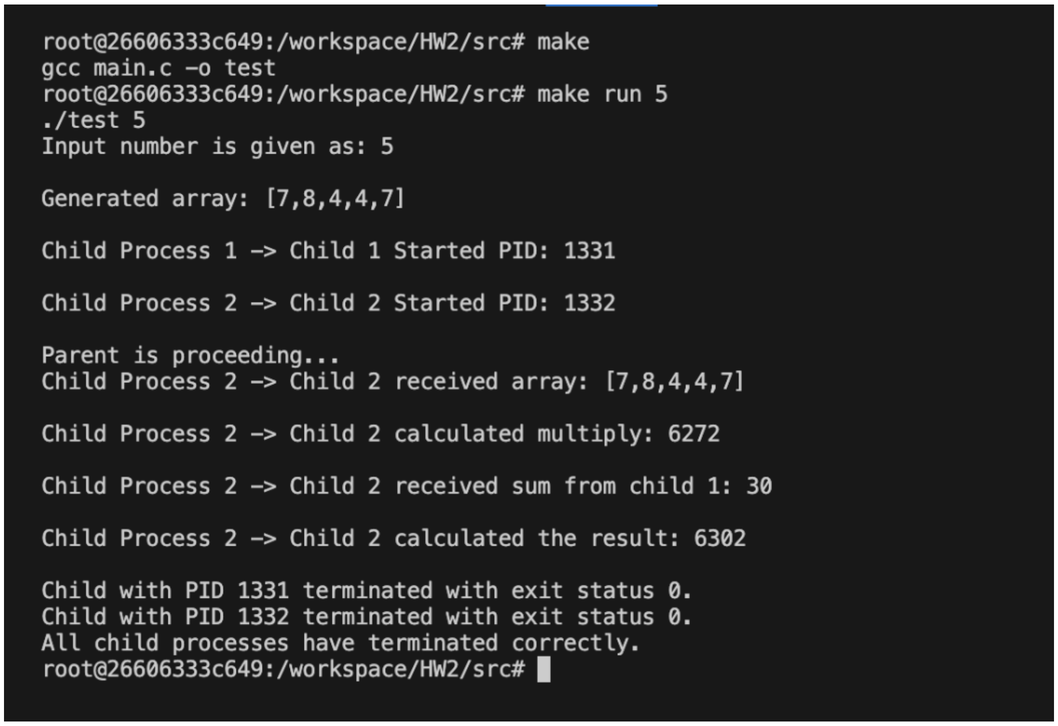


```c
while (children > 0) {
    printf("Parent is proceeding...\n");
    sleep(2);
}

if (children != 0) {
    fprintf(stderr, "Error: Child process count mismatch. Expected %d, got %d.\n", 0, children);
} else {
    printf("All child processes have terminated correctly.\n");
}
```

The parent process waits in a while loop until all child processes have terminated. Once the number of active children reaches zero, indicating that all children have finished, the parent process also terminates.

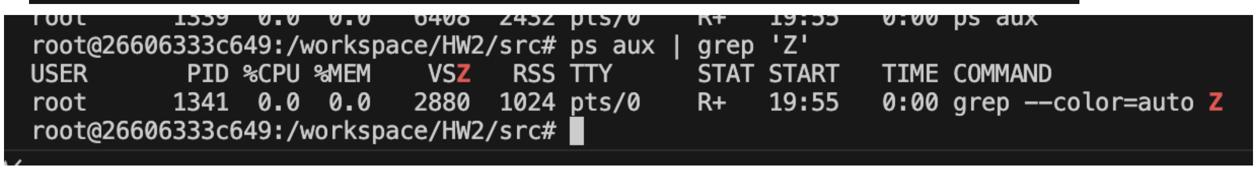
Test 1

Output



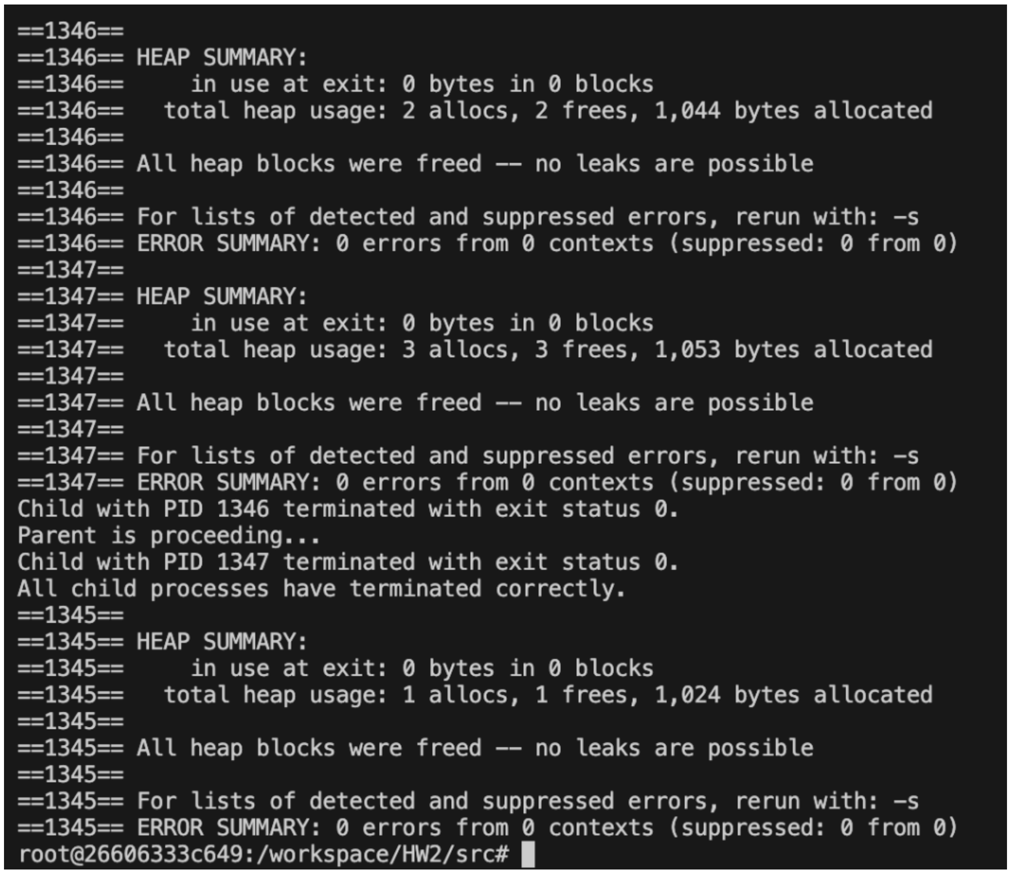
```
root@26606333c649:/workspace/HW2/src# make
gcc main.c -o test
root@26606333c649:/workspace/HW2/src# make run 5
./test 5
Input number is given as: 5

Generated array: [7,8,4,4,7]

Child Process 1 -> Child 1 Started PID: 1331

Child Process 2 -> Child 2 Started PID: 1332

Parent is proceeding...
Child Process 2 -> Child 2 received array: [7,8,4,4,7]

Child Process 2 -> Child 2 calculated multiply: 6272

Child Process 2 -> Child 2 received sum from child 1: 30

Child Process 2 -> Child 2 calculated the result: 6302

Child with PID 1331 terminated with exit status 0.
Child with PID 1332 terminated with exit status 0.
All child processes have terminated correctly.
root@26606333c649:/workspace/HW2/src# 
```

Zombie test



```
root      1339  0.0  0.0   6408  2432 pts/0    R+   19:55   0:00 ps aux
root@26606333c649:/workspace/HW2/src# ps aux | grep 'Z'
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root      1341  0.0  0.0   2880  1024 pts/0    R+   19:55   0:00 grep --color=auto Z
root@26606333c649:/workspace/HW2/src# 
```

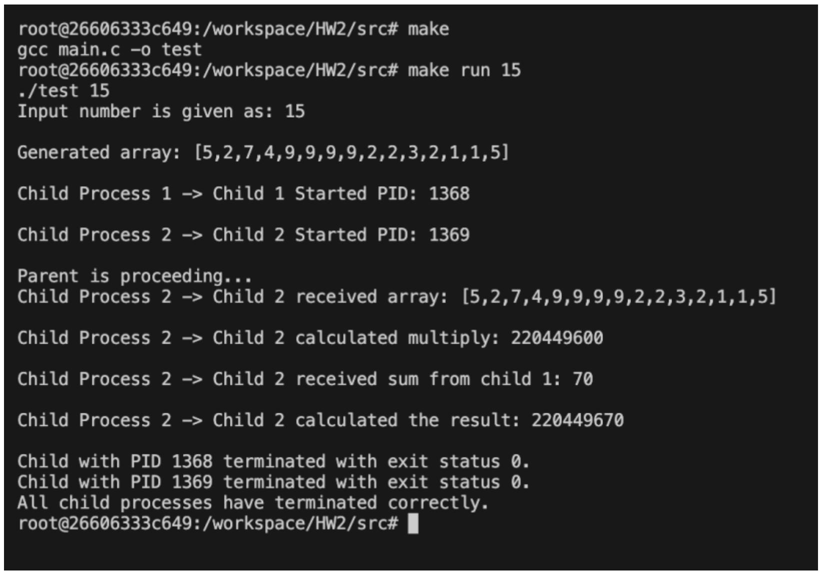
Memory Leak test



```
==1346==
==1346== HEAP SUMMARY:
==1346==     in use at exit: 0 bytes in 0 blocks
==1346==   total heap usage: 2 allocs, 2 frees, 1,044 bytes allocated
==1346==
==1346== All heap blocks were freed -- no leaks are possible
==1346==
==1346== For lists of detected and suppressed errors, rerun with: -s
==1346== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==1347==
==1347== HEAP SUMMARY:
==1347==     in use at exit: 0 bytes in 0 blocks
==1347==   total heap usage: 3 allocs, 3 frees, 1,053 bytes allocated
==1347==
==1347== All heap blocks were freed -- no leaks are possible
==1347==
==1347== For lists of detected and suppressed errors, rerun with: -s
==1347== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
Child with PID 1346 terminated with exit status 0.
Parent is proceeding...
Child with PID 1347 terminated with exit status 0.
All child processes have terminated correctly.
==1345==
==1345== HEAP SUMMARY:
==1345==     in use at exit: 0 bytes in 0 blocks
==1345==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==1345==
==1345== All heap blocks were freed -- no leaks are possible
==1345==
==1345== For lists of detected and suppressed errors, rerun with: -s
==1345== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
root@26606333c649:/workspace/HW2/src#
```

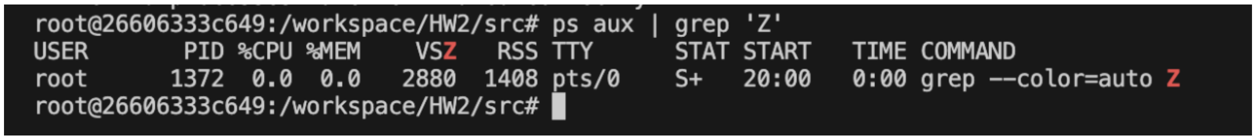
Test2

Output



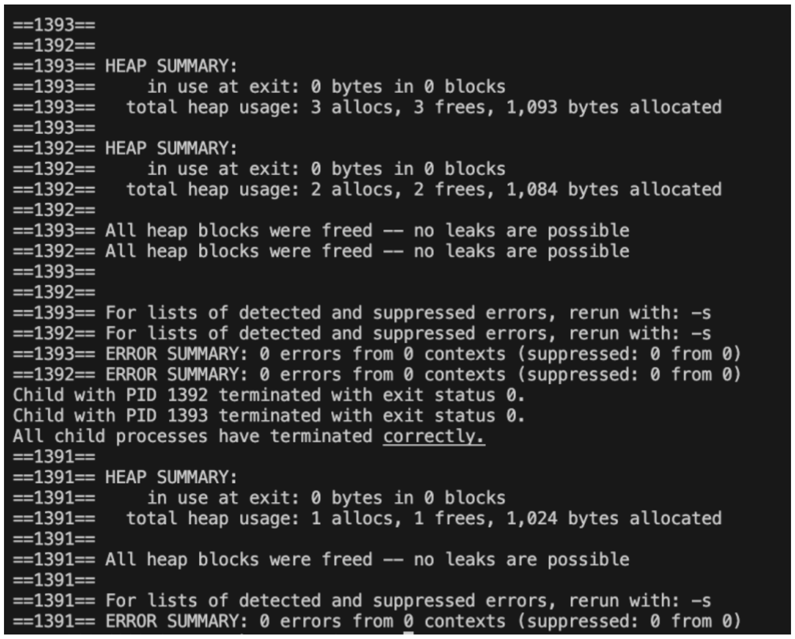
```
root@26606333c649:/workspace/HW2/src# make
gcc main.c -o test
root@26606333c649:/workspace/HW2/src# make run 15
./test 15
Input number is given as: 15

Generated array: [5,2,7,4,9,9,9,9,2,2,3,2,1,1,5]

Child Process 1 -> Child 1 Started PID: 1368

Child Process 2 -> Child 2 Started PID: 1369

Parent is proceeding...
Child Process 2 -> Child 2 received array: [5,2,7,4,9,9,9,9,2,2,3,2,1,1,5]

Child Process 2 -> Child 2 calculated multiply: 220449600

Child Process 2 -> Child 2 received sum from child 1: 70

Child Process 2 -> Child 2 calculated the result: 220449670

Child with PID 1368 terminated with exit status 0.
Child with PID 1369 terminated with exit status 0.
All child processes have terminated correctly.
root@26606333c649:/workspace/HW2/src# 
```

Zombie



```
root@26606333c649:/workspace/HW2/src# ps aux | grep 'Z'
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root      1372  0.0  0.0   2880  1408 pts/0    S+   20:00   0:00 grep --color=auto Z
root@26606333c649:/workspace/HW2/src# 
```

Memory Leak



```
==1393==
==1392==
==1393== HEAP SUMMARY:
==1393==     in use at exit: 0 bytes in 0 blocks
==1393==   total heap usage: 3 allocs, 3 frees, 1,093 bytes allocated
==1393==
==1392== HEAP SUMMARY:
==1392==     in use at exit: 0 bytes in 0 blocks
==1392==   total heap usage: 2 allocs, 2 frees, 1,084 bytes allocated
==1392==
==1393== All heap blocks were freed -- no leaks are possible
==1392== All heap blocks were freed -- no leaks are possible
==1393==
==1392==
==1393== For lists of detected and suppressed errors, rerun with: -s
==1392== For lists of detected and suppressed errors, rerun with: -s
==1393== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==1392== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
Child with PID 1392 terminated with exit status 0.
Child with PID 1393 terminated with exit status 0.
All child processes have terminated correctly.
==1391==
==1391== HEAP SUMMARY:
==1391==     in use at exit: 0 bytes in 0 blocks
==1391==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==1391==
==1391== All heap blocks were freed -- no leaks are possible
==1391==
==1391== For lists of detected and suppressed errors, rerun with: -s
==1391== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Additional Information

All tests were conducted using Docker. It was also re-tested on a computer with Ubuntu 22.04 installed. Just in case, I'm including the Docker file inside the document as well.