CS 332/532 Systems Programming

Lecture 25
I/O Redirection

Professor : Mahmut Unan – UAB CS

Agenda

- Review Linux I/O Streams
- Sharing between parent and child processes

HW4

Exam 2

- Exam: Friday 11/01/2024
  - Lecture time
  - Canvas Quiz & Lockdown browser
  - in class, on computer
- Similar to Exam 1
- Wednesday - review

Linux I/O Streams

- Before we discuss I/O redirection, let us review how I/O is handled in Linux systems.

- Each process in a Linux environment has three different file descriptors available when a process is created: standard input (*stdin – 0*), standard output (*stdout – 1*), and standard error (*stderr – 2*).

- These three file descriptors are created when a process is created.

- We use the *stdin* file descriptor to read input from a keyboard or from another a file or from another program.

- Similarly, we use the *stdout* and *stderr* file descriptors to write output and error messages, respectively, to the terminal.

- Input and output in the Linux environment is distributed across three streams. These streams are:
- **standard input** (**stdin**)
- **standard output** (**stdout**)
- **standard error** (**stderr**)

- The streams are also numbered:
- **stdin** (**0**)
- **stdout** (**1**)
- **stderr** (**2**)

During standard interactions between the user and the terminal, standard input is transmitted through the user's keyboard. Standard output and standard error are displayed on the user's terminal as text. Collectively, the three streams are referred to as the *standard streams*.

cat command

```
(base) mahmutunan@MacBook-Pro lecture26 % cat
1
1
2
2
3
3
(base) mahmutunan@MacBook-Pro lecture26 %
```

Stream Redirection

- **Overwrite**

    > - standard output

    < - standard input

    2> - standard error

- **Append**

    >> - standard output

    << - standard input

    2>> - standard error

```
(base) mahmutunan@MacBook-Pro lecture26 % cat > outputFile.txt
1
2
3
4
(base) mahmutunan@MacBook-Pro lecture26 % cat outputFile.txt
1
2
3
4
```

```
(base) mahmutunan@MacBook-Pro lecture26 % cat >> outputFile.txt
a
b
c
(base) mahmutunan@MacBook-Pro lecture26 % cat outputFile.txt
1
2
3
4
a
b
c
```

```
(base) mahmutunan@MacBook-Pro lecture26 % ls >> listOffiles.txt
(base) mahmutunan@MacBook-Pro lecture26 % cat listOffiles.txt
error.txt
forkexecvp
forkexecvp.c
forkexecvp2.c
hw2.c
input.txt
ioredirect.c
lab7_solution.c
listOffiles.txt
myprog
myprog.c
output.txt
output2.txt
outputFile.txt
```

stdout stderr

```
(base) mahmutunan@MacBook-Pro lecture26 % echo "some output using stdout"
some output using stdout
(base) mahmutunan@MacBook-Pro lecture26 % echo

(base) mahmutunan@MacBook-Pro lecture26 % _
```

```
(base) mahmutunan@MacBook-Pro lecture26 % cat nonexistingfile.txt
cat: nonexistingfile.txt: No such file or directory
(base) mahmutunan@MacBook-Pro lecture26 % _
```

- You have already been using these file descriptors when you wrote the insertion sort program in C.

- You read the number of elements and the elements to be sorted from the keyboard using the *scanf* function.

- The *scanf* function was using *stdin* file descriptor to read your keyboard input. In other words, the following two functions are equivalent:

```
scanf("%d", &N);
fscanf(stdin, "%d", &N);
```

- Similarly when you use the *printf* function to print the output of your program you are using the *stdout* file descriptor.

- The two functions below are equivalent:

```
printf("%d\n", N);
fprintf(stdout,"%d\n", N);
```

- The file descriptors *stdin, stdout,* and *stderr* are defined in the header file *stdio.h*. We typically use the *stderr* file descriptor to write error messages.

- If we need to save the output or error message from a program to a file or read data from a file instead of entering it through the keyboard, we can use the I/O redirection supported by the Linux shell (such as bash).

- In fact, we already used this in one of the earlier labs when we used insertion sort to sort large input values.

- The following examples show how to use I/O redirection in the bash shell to read input from a file (instead of entering it from the keyboard) and send the output and error messages to different files (instead of the terminal)

Exercise 1

```c
#include <stdio.h>

int main(int argc, char **argv) {
    char name[BUFSIZ];

    printf("Please enter your name: ");
    scanf("%s", name);
    printf("[stdout]: Hello %s!\n", name);
    fprintf(stderr, "[stderr]: Hello %s!\n", name);

    return 0;
}
```

Compile the program myprog.c,
create a file called *input.txt*, type you name in the file *input.txt*

compile & run

```
[(base) mahmutunan@MacBook-Pro lecture26 % touch input.txt
[(base) mahmutunan@MacBook-Pro lecture26 % echo "Mahmut" > input.txt
[(base) mahmutunan@MacBook-Pro lecture26 % cat input.txt
Mahmut
[(base) mahmutunan@MacBook-Pro lecture26 % gcc -Wall myprog.c -o myprog
[(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt
Please enter your name: [stdout]: Hello Mahmut!
[stderr]: Hello Mahmut!
[(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt > output.txt
[stderr]: Hello Mahmut!
(base) mahmutunan@MacBook-Pro lecture26 % _
```

📄 input.txt

```
Mahmut
```

📄 output.txt

```
Please enter your name: [stdout]: Hello Mahmut!
```
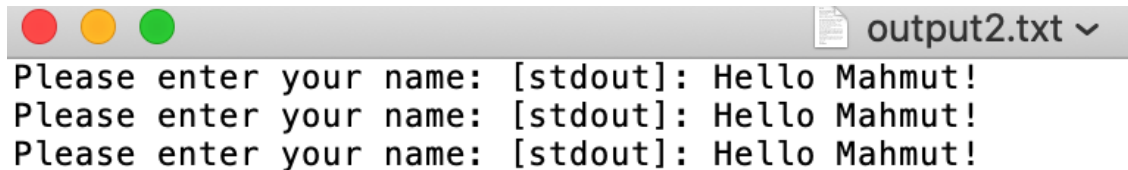
16
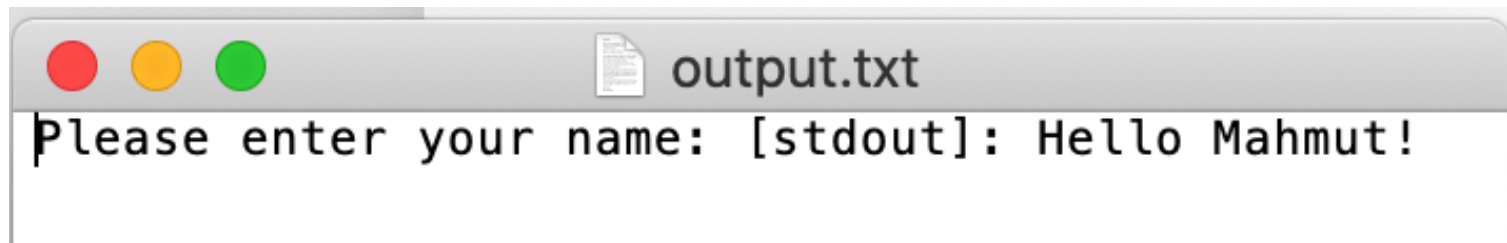
compile & run

```
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt 2> error.txt
Please enter your name: [stdout]: Hello Mahmut!
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt 2> error.txt >output.txt
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt >output2.txt 2> error.txt
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt & >output2.txt 2> error.txt
[1] 91593
Please enter your name: [stdout]: Hello Mahmut!
[stderr]: Hello Mahmut!
[1]  + done       ./myprog < input.txt
^C
(base) mahmutunan@MacBook-Pro lecture26 % _
```

📄 error.txt

Please enter your name: [stdout]: Hello Mahmut!

compile & run

```
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt >> output2.txt
[stderr]: Hello Mahmut!
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt >> output2.txt
[stderr]: Hello Mahmut!
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt >> output2.txt
[stderr]: Hello Mahmut!
(base) mahmutunan@MacBook-Pro lecture26 % _
```

output2.txt

```
Please enter your name: [stdout]: Hello Mahmut!
Please enter your name: [stdout]: Hello Mahmut!
Please enter your name: [stdout]: Hello Mahmut!
```

You can replace **>** with **>>** if you like to append to the file
 instead of overwriting the file

18

Sharing between parent and child processes

- When we created a new process using fork/exec in the previous labs, we noted that the child process is a copy of the parent process and it inherits several attributes from the parent process such as open file descriptors.

- This duplication of descriptors allowed the child processes to read and write to the standard I/O streams (note that the child process was able to read input from the keyboard and write output to the terminal).

- As a result of this sharing both parent and child processes share the three standard I/O streams: *stdin*, *stdout*, and *stderr*.

Exercise 2

- Let us use the example from the previous lectures to illustrate this by adding the following lines in the parent process:

```
char name[BUFSIZ];

    printf("Please enter your name: ");
    scanf("%s", name);
    printf("[stdout]: Hello %s!\n", name);
    fprintf(stderr, "[stderr]: Hello
%s!\n", name);
```

Exercise 2

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char **argv) {
    pid_t pid;
    int status;

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

    pid = fork();
    if (pid == 0) { /* this is child process */
        execvp(argv[1], &argv[1]);
    perror("exec");
    exit(-1);
    } else if (pid > 0) { /* this is the parent process */
```

Exercise 2

```
23              char name[BUFSIZ];
24
25              printf("[%d]: Please enter your name: ", getpid());
26              scanf("%s", name);
27              printf("[stdout]: Hello %s!\n",  name);
28              fprintf(stderr, "[stderr]: Hello %s!\n",  name);
29
30              wait(&status); /* wait for the child process to terminate */
31              if (WIFEXITED(status)) { /* child process terminated normally */
32                  printf("Child process exited with status = %d\n", WEXITSTATUS(status));
33              } else { /* child process did not terminate normally */
34                  printf("Child process did not terminate normally!\n");
35                  /* look at the man page for wait (man 2 wait) to determine
36                     how the child process was terminated */
37              }
38          } else { /* we have an error */
39              perror("fork"); /* use perror to print the system error message */
40              exit(EXIT_FAILURE);
41          }
42
43          return 0;
44      }
45
```

compile & run

- If we compile and execute the program by using *myprog* (used earlier) as the child process, we will notice that the prompt to enter the name is printed twice. If we enter the name, which process is reading the keyboard input?

```
(base) mahmutunan@MacBook-Pro lecture26 % gcc -Wall forkexecvp2.c -o forkexecvp
(base) mahmutunan@MacBook-Pro lecture26 % ./forkexecvp ./myprog
[86530]: Please enter your name: Please enter your name: mahmut
[stdout]: Hello mahmut!
[stderr]: Hello mahmut!
```

Exercise 2

```c
        char name[BUFSIZ];

        printf("[%d]: Please enter your name: ", getpid());
        scanf("%s", name);
        printf("[%d-stdout]: Hello %s!\n", getpid(), name);
        fprintf(stderr, "[%d-stderr]: Hello %s!\n", getpid(), name);

        wait(&status); /* wait for the child process to terminate */
        if (WIFEXITED(status)) { /* child process terminated normally */
            printf("Child process exited with status = %d\n", WEXITSTATUS(status));
        } else { /* child process did not terminate normally */
            printf("Child process did not terminate normally!\n");
            /* look at the man page for wait (man 2 wait) to determine
               how the child process was terminated */
        }
    } else { /* we have an error */
        perror("fork"); /* use perror to print the system error message */
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

compile & run

- We could add the PID in the printf statement to make this explicit. We can update the code above to print the PID and test the program. In any case, this illustrates the result of sharing of the standard I/O streams between the parent and the child processes

```
(base) mahmutunan@MacBook-Pro lecture26 % gcc -Wall forkexecvp2.c -o forkexecvp
(base) mahmutunan@MacBook-Pro lecture26 % ./forkexecvp ./myprog
[85953]: Please enter your name: Please enter your name: mahmut
[85953-stdout]: Hello mahmut!
[85953-stderr]: Hello mahmut!
```

- If we like to change the behavior of all child processes to use separate files instead of the standard I/O streams we have to replace the standard I/O file descriptors with new file descriptors.

- We will use the dup2() system call to create a copy of this file descriptors and associate separate files to replace the standard I/O streams.

dup2()

- **Copying file descriptors – dup2() system call**
- The dup2() system call duplicates an existing file descriptor and returns the duplicate file descriptor.
- After the call returns successfully, both file descriptors can be used interchangeably. If there is an error then -1 is returned and the corresponding errno is set (look at the man page for dup2() for more details on the specific error codes returned).
- The prototype for the dup2() system call is shown below:

```
#include <unistd.h>

int dup2(int oldfd, int newfd);
```

dup() vs dup2()

- These system calls create a copy of the file descriptor *oldfd*.
- **dup**() uses the lowest-numbered unused descriptor for the new descriptor.
- **dup2**() makes *newfd* be the copy of *oldfd*, closing *newfd* first if necessary, but note the following:
  - If *oldfd* is not a valid file descriptor, then the call fails, and *newfd* is not closed.
  - If *oldfd* is a valid file descriptor, and *newfd* has the same value as *oldfd*, then **dup2**() does nothing, and returns *newfd*.
- After a successful return from one of these system calls, the old and new file descriptors may be used interchangeably. T