National University
of computer and emerging sciences

CL 2006 OPERATING SYSTEM

BS Software Engineering

Fall-2024

Course Instructors: **Syed Daniyal Hussain Shah, Hifza Umar**

Course Coordinator: **Syed Daniyal Hussain Shah**

Lab Policy and Rules:

1. **100% attendance is mandatory.** In case of an emergency, you can avail yourself of up to 3 absences. So, don't waste your absences , save them for emergencies. If you are debarred from the course due to low attendance, do not come to me to correct it.

2**. Disturbing the class environment during lab sessions.** such as by talking, using mobile phones, eating, etc. will result in penalties (e.g., deduction of marks).

3. **Lab tasks will not be accepted if you are absent.** If you miss a lab, you cannot submit that lab task.

4**. Lab demos for each lab task will be conducted at the end of each session.** If you miss the demo for any reason, no retake will be allowed, and that lab will be marked as 0.

5. **All quizzes will be unannounced**. Be prepared for surprise quizzes. Quizzes will cover content from previous labs as well as theory lectures from the corresponding theory course.

6. **You can take help from the internet for lab tasks,** but simply copying and pasting without understanding will be marked as 0. You should be able to explain the syntax or material used in your tasks.

7**. Do not ask for personal favors.** If you have concerns, such as short attendance, please speak with the relevant authority (e.g., academics).

8**. Students on warning:** Now is the time to study and earn a good grade. Do not ask for extra marks at the end if you are unable to clear your warning.

Lab 6: Inter-process Communication

File System Call

Open():

Opens a file and returns a file descriptor that can be used for subsequent operations like reading, writing, etc.

```
int open(const char *pathname, int flags, mode_t mode);
```

pathname: Path to the file.

flags: Access modes like O_RDONLY (read-only), O_WRONLY (write-only), O_RDWR (read and write).

mode: Optional, used for setting permissions when creating a file (e.g., 0644).

Three standard file descriptors,

1: standard input

0: standard output

2: standard error

Read():

Reads data from a file descriptor into a buffer.

```
ssize_t read(int fd, void *buf, size_t count);
```

fd: File descriptor returned by open().

buf: Buffer to store the data.

count: Number of bytes to read..

Write():

Writes data from a buffer to a file descriptor.

```
ssize_t write(int fd, const void *buf, size_t count);
```

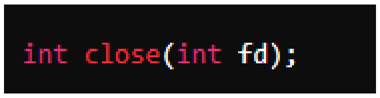
fd: File descriptor returned by open().

buf: Buffer containing the data to write.

count: Number of bytes to write.

Close():

Closes a file descriptor, freeing the resources associated with it.



```
int close(int fd);
```

Example 1: Using open(), read(), and close() to Read a File:

```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

int main() {
    int fd = open("example.txt", O_RDONLY);
    if (fd == -1) {
        printf("Failed to open the file.\n");
        return 1;
    }

    char buffer[100];
    size_t bytesRead = read(fd, buffer, sizeof(buffer) - 1);
    if (bytesRead == -1) {
        printf("Failed to read from the file.\n");
        close(fd);
        return 1;
    }
    buffer[bytesRead] = '\0';

    printf("File contents:\n%s\n", buffer);

    close(fd);

    return 0;

}
```

- #include <fcntl.h>: This header file is required for the open() system call, which is used to open files in Linux.
- #include <unistd.h>: This header file provides the declarations for the read() and close() system calls, which are used to read from and close file descriptors.
- int fd = open("example.txt", O_RDONLY);: This opens the file example.txt in read-only mode. O_RDONLY specifies that the file is to be opened for reading only.
- If the file cannot be opened (e.g., if it doesn't exist or there are insufficient permissions), open() returns -1. The program checks for this with if (fd == -1) and prints an error message, then exits with a return value of 1 to indicate failure.
- char buffer[100];: A buffer (array of characters) is declared to hold the data read from the file.
- ssize_t bytesRead = read(fd, buffer, sizeof(buffer) - 1);: The read() system call reads data from the file descriptor fd into the buffer. It reads up to sizeof(buffer) - 1 bytes (i.e., 99 bytes) to leave space for a null terminator at the end of the string. The number of bytes actually read is stored in bytesRead.
- buffer[bytesRead] = '\0';: The buffer is null-terminated by adding a null character ('\0') at the position after the last byte read.

Example 2: Using open(), write(), and close() to Read a File:

```
#include <fcntl.h>

#include <unistd.h>

#include <stdio.h>
```

```
int main() {
    int fd = open("output.txt", O_WRONLY | O_CREAT, 0644);
    if (fd == -1) {
        printf("Failed to open the file.\n");
        return 1;
    }
    const char *message = "Hello, world!\n";
    ssize_t bytesWritten = write(fd, message, 14);
    if (bytesWritten == -1) {
        printf("Failed to write to the file.\n");
        close(fd);
        return 1;
    }
    printf("Data written successfully.\n");
    close(fd);
    return 0;
}
```

- int fd = open("output.txt", O_WRONLY | O_CREAT, 0644);: Opens or creates the file output.txt for writing.
- O_WRONLY specifies that the file is opened in write-only mode.
- O_CREAT creates the file if it does not already exist.
- 0644 sets the file permissions to rw-r--r-- (owner can read/write, group and others can read). The leading 0 indicates that the permissions are specified in octal format.
- If open() fails, it returns -1.
- ssize_t bytesWritten = write(fd, message, 14);: Writes 14 bytes from the message string to the file descriptor fd. write() returns the number of bytes written, which should match the size of the message if successful.

Inter-Process Communication:

Inter-process communication (IPC) in Linux allows processes to communicate with each other. One of the simplest forms of IPC is using unnamed pipes. Unnamed pipes are a way to allow data to be passed from one process to another in a unidirectional manner. They are often used in scenarios where you want to establish a communication channel between a parent and a child process.

A process contains everything in its address space needed for the execution. Processes can be independent or cooperating processes.

Independent Processes:

Processes operating concurrently on a systems are those that can neither affect other processes or be affected by other processes.

Cooperating Processes:

Processes that can affect or be affected by other processes. Processes may wish to communicate with each other in order to share some data.

Pipes:

One of the IPC mechanism that allow related-processes to communicate is the pipe. Also called anonymous or unnamed pipe.

Pipe allows related processes to (parent- Child) to send a data from one process to other process.

Pipe() System call:

An unnamed pipe is created using the `pipe()` system call, which takes an array of two file descriptors as an argument. The first file descriptor refers to the read end of the pipe, and the second refers to the write end.

Data written to the write end of the pipe can be read from the read end. The data flow is unidirectional, meaning you can only read from the read end and write to the write end.

After creating a pipe, the file descriptors can be used with standard I/O operations such as read() and write().

Example 1:

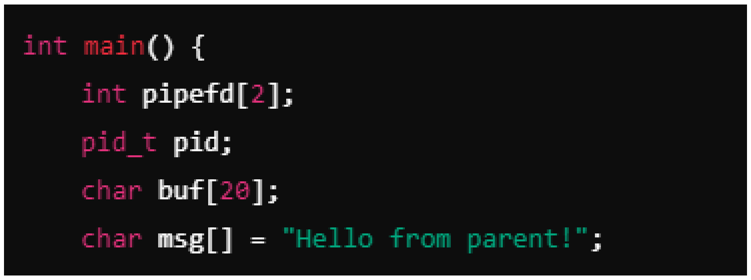
```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>
```

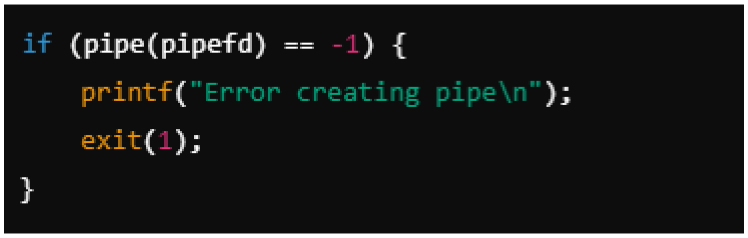
```
#include <string.h>
int main() {
    int pipefd[2];
    pid_t pid;
    char buf[20];
    char msg[] = "Hello from parent!";
    if (pipe(pipefd) == -1) {
        printf("Error creating pipe\n");
        exit(1);
    }
    pid = fork();
    if (pid == -1) {
        printf("Error creating process\n");
        exit(2);
    }
    if (pid == 0) {
        close(pipefd[1]);
        ssize_t bytesRead = read(pipefd[0], buf, sizeof(buf));
        if (bytesRead == -1) {
            printf("Error reading from pipe\n");
            exit(3);
        }
        printf("Child read: %s\n", buf);
        close(pipefd[0]);
        exit(0);
    } else {
        close(pipefd[0]);
        ssize_t bytesWritten = write(pipefd[1], msg, sizeof(msg));
        if (bytesWritten == -1) {
            printf("Error writing to pipe\n");
            exit(4);
        }
        close(pipefd[1]);
        wait(NULL);
        exit(0);
    }
}
```



```
int main() {
    int pipefd[2];
    pid_t pid;
    char buf[20];
    char msg[] = "Hello from parent!";
```

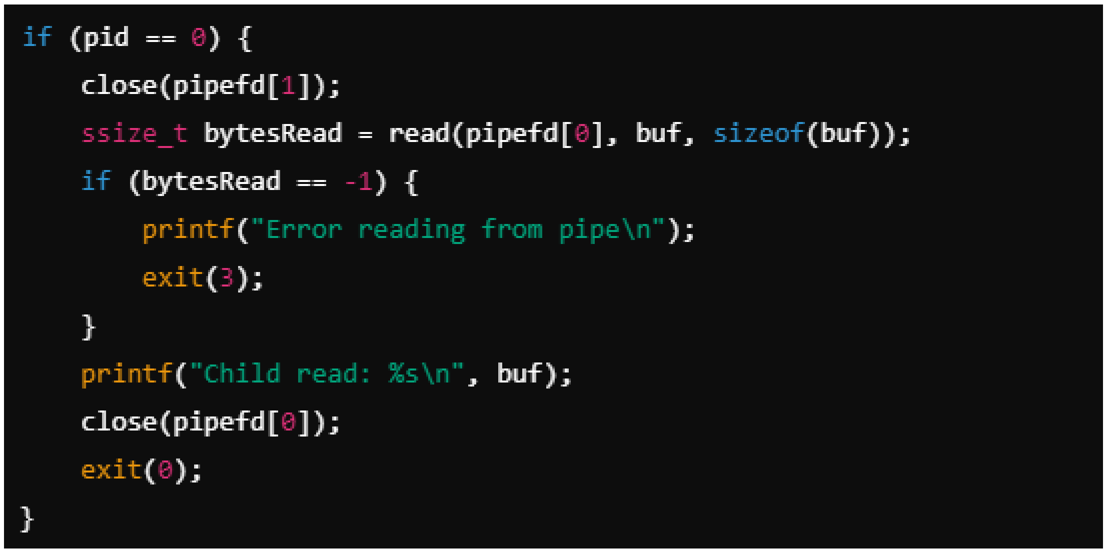
File Descriptors Array: pipefd[2] holds the file descriptors for the pipe. pipefd[0] is the read end, and pipefd[1] is the write end.

Buffers: buf is used to store data read from the pipe, and msg contains the message to be sent.



```
if (pipe(pipefd) == -1) {
    printf("Error creating pipe\n");
    exit(1);
}
```

pipe(pipefd) creates a pipe. If it fails, an error message is printed, and the program exits with status code 1 else it returns 0.



```
if (pid == 0) {
    close(pipefd[1]);
    ssize_t bytesRead = read(pipefd[0], buf, sizeof(buf));
    if (bytesRead == -1) {
        printf("Error reading from pipe\n");
        exit(3);
    }
    printf("Child read: %s\n", buf);
    close(pipefd[0]);
    exit(0);
}
```

Close Write End: close(pipefd[1]) closes the unused write end of the pipe.

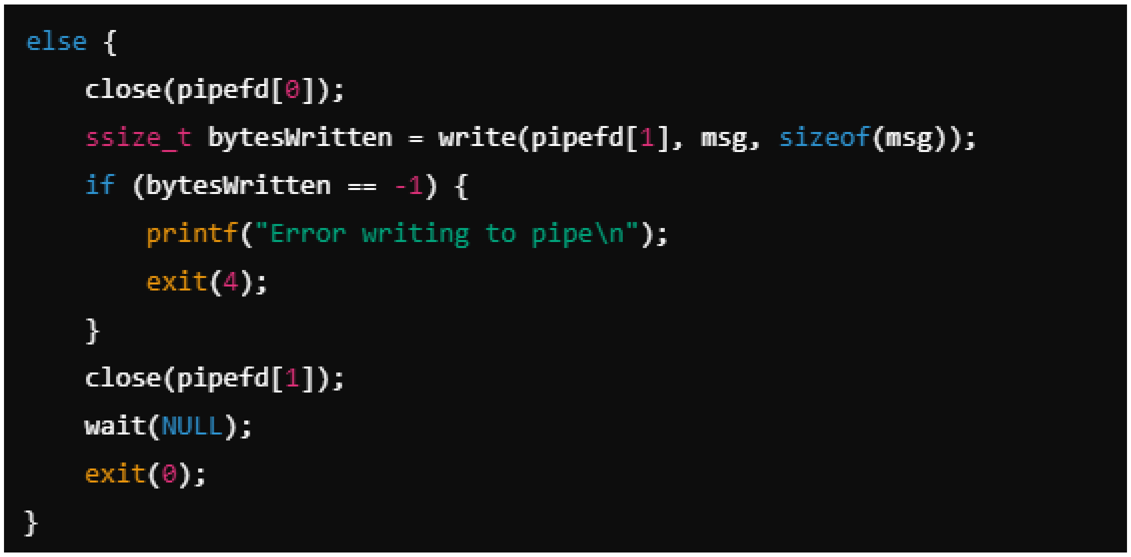
Read from Pipe: read(pipefd[0], buf, sizeof(buf)) reads data from the read end of the pipe into buf.

If reading fails, it prints an error message and exits with status code 3.

Print Data: printf("Child read: %s\n", buf) prints the data read from the pipe.

Close Read End: close(pipefd[0]) closes the read end of the pipe.

Exit: exit(0) exits the child process successfully.



```
else {
    close(pipefd[0]);
    ssize_t bytesWritten = write(pipefd[1], msg, sizeof(msg));
    if (bytesWritten == -1) {
        printf("Error writing to pipe\n");
        exit(4);
    }
    close(pipefd[1]);
    wait(NULL);
    exit(0);
}
```

Close Read End: close(pipefd[0]) closes the unused read end of the pipe.

Write to Pipe: write(pipefd[1], msg, sizeof(msg)) writes the message to the write end of the pipe.

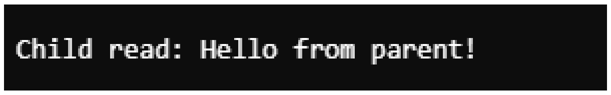
If writing fails, it prints an error message and exits with status code 4.

Close Write End: close(pipefd[1]) closes the write end of the pipe.

Wait for Child: wait(NULL) waits for the child process to terminate.

Exit: exit(0) exits the parent process successfully.

Output:



```
Child read: Hello from parent!
```

Example 2:

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

#include <string.h>


int main() {

    int pipe1[2];

    int pipe2[2];

    pid_t pid;

    char parent_msg[] = "Hello from parent!";

    char child_msg[] = "Hello from child!";

    char buf[100];


    if (pipe(pipe1) == -1) {

        printf("Error creating pipe1\n");

        exit(1);

    }
```

```
if (pipe(pipe2) == -1) {
    printf("Error creating pipe2\n");
    exit(2);
}
pid = fork();
if (pid == -1) {
    printf("Error creating process\n");
    exit(3);
}
if (pid == 0) {
    close(pipe1[1]);
    close(pipe2[0]);
    ssize_t bytesRead = read(pipe1[0], buf, sizeof(buf) - 1);
    if (bytesRead == -1) {
        printf("Error reading from pipe1\n");
        exit(4);
    }
    buf[bytesRead] = '\0';
    printf("Child received: %s\n", buf);
    ssize_t bytesWritten = write(pipe2[1], child_msg, sizeof(child_msg));
    if (bytesWritten == -1) {
        printf("Error writing to pipe2\n");
        exit(5);
    }
    close(pipe1[0]);
    close(pipe2[1]);
    exit(0);
} else {
    close(pipe1[0]);
```

```
        close(pipe2[1]);

        ssize_t bytesWritten = write(pipe1[1], parent_msg, sizeof(parent_msg));

        if (bytesWritten == -1) {

            printf("Error writing to pipe1\n");

            exit(6);

        }

        ssize_t bytesRead = read(pipe2[0], buf, sizeof(buf) - 1);

        if (bytesRead == -1) {

            printf("Error reading from pipe2\n");

            exit(7);

        }

        buf[bytesRead] = '\0';

        printf("Parent received: %s\n", buf);

        close(pipe1[1]);

        close(pipe2[0]);

        wait(NULL);

        exit(0);

    }

}
```

Pipe Creation:

`pipe1[2]` is used for communication from the parent to the child (parent writes, child reads).

`pipe2[2]` is used for communication from the child to the parent (child writes, parent reads).

Child Process:

Closes the unused write end of pipe1 and the unused read end of pipe2.

Reads the message from pipe1 sent by the parent.

Writes a response to pipe2 for the parent.

Parent Process:

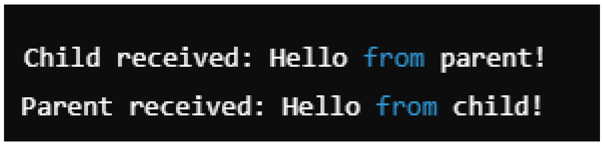
Closes the unused read end of pipe1 and the unused write end of pipe2.

Writes a message to pipe1 for the child.

Reads the response from pipe2 sent by the child.

The parent process waits for the child process to finish using `wait(NULL)`.

Output:



```
Child received: Hello from parent!
Parent received: Hello from child!
```

Class activity:

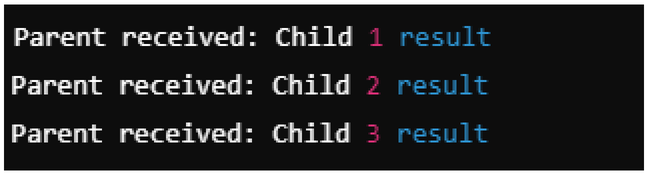
Parent Process will create a pipe using file descriptor array and pipe system call. Then Parent Process will create a child process using the fork system call. After that using code(written in the condition body) which is relevant with Parent Process, Parent process will WRITE a int type number stored in the variable on a pipe(already created). At last using code(written in the condition body) which is relevant with Child Process, Child process will READ that number from the pipe into a int type variable and then child process will check that number whether it is ODD or EVEN number. And display the message “Number is ODD” or “Number is EVEN”.

Tasks:

I. Write a program that takes two command-line arguments: an input file and an output file. The program should copy the contents of the input file to the output file using read() and write() system calls. Make sure to handle errors if the files cannot be opened. **[Estimated Time: 15 mins]**

II. Extend the example 2 to allow multiple messages to be sent back and forth between the parent and child processes.
Implement a program where the parent process sends a series of three messages to the child process through one pipe.
The child process should read each message, print it, and then send a response message back to the parent process through another pipe.
Both processes should handle multiple messages in a loop and ensure that each message is correctly sent and received. **[Estimated Time: 30 mins]**

III. Create multiple child processes, each communicating with the parent through shared pipes.
Implement a parent process that spawns three child processes.
Each child process should perform some simple computation (e.g., compute the square of a number) and send the result to the parent through a pipe.
The parent process should collect results from all child processes and print them.
[Estimated Time: 45 mins]



```
Parent received: Child 1 result
Parent received: Child 2 result
Parent received: Child 3 result
```

Task Submission Guidelines:

1. **Make a word document and paste the code with output's screenshot there and save it as pdf or odt.**
2. **Include your name and roll no. at the front page.**
3. **Files other than pdf or odt will not be accepted and will be marked as 0.**