Pipes

What are pipes in OS?

- A pipe simply refers to a temporary software connection between two programs or commands for **IPC**
- An area of the main memory is treated like a virtual file to temporarily hold data and pass it from one process to another in a single direction
- In OSes like Unix, a pipe passes the output of one process to another process

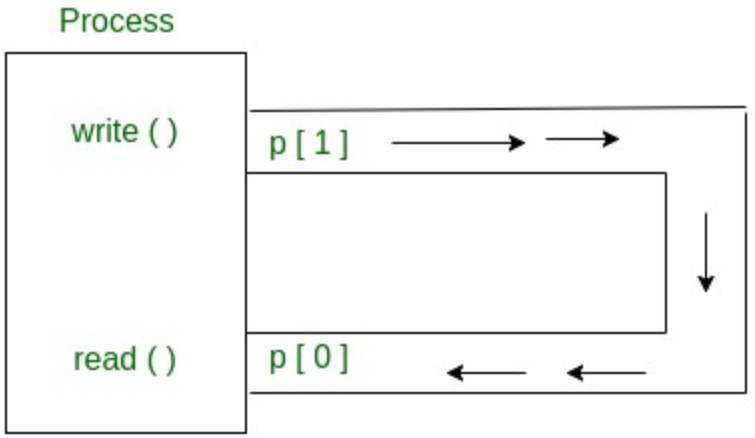
```
int pipe(int fds[2]);

Parameters :
fd[0] will be the fd(file descriptor) for the read end of pipe.
fd[1] will be the fd for the write end of pipe.
Returns : 0 on Success.
-1 on error.
```

pipes

- Pipe is one-way communication only i.e we can use a pipe such that One process writes to the pipe, and the other process reads from the pipe
- The pipe can be used by the creating process, as well as all its child processes, for reading and writing. One process can write to **this "virtual file" or pipe** and another related process can read from it.
- If a process tries to read before something is written to the pipe, the process is **suspended** until something is written.
- The pipe system call finds the first two available positions in the process's open file table and allocates them for the read and write ends of the pipe.

- Pipes behave FIFO(First in First out),i.e., like a queue data structure
- Size of read and write don't have to match here



Example: what is the output of this program?

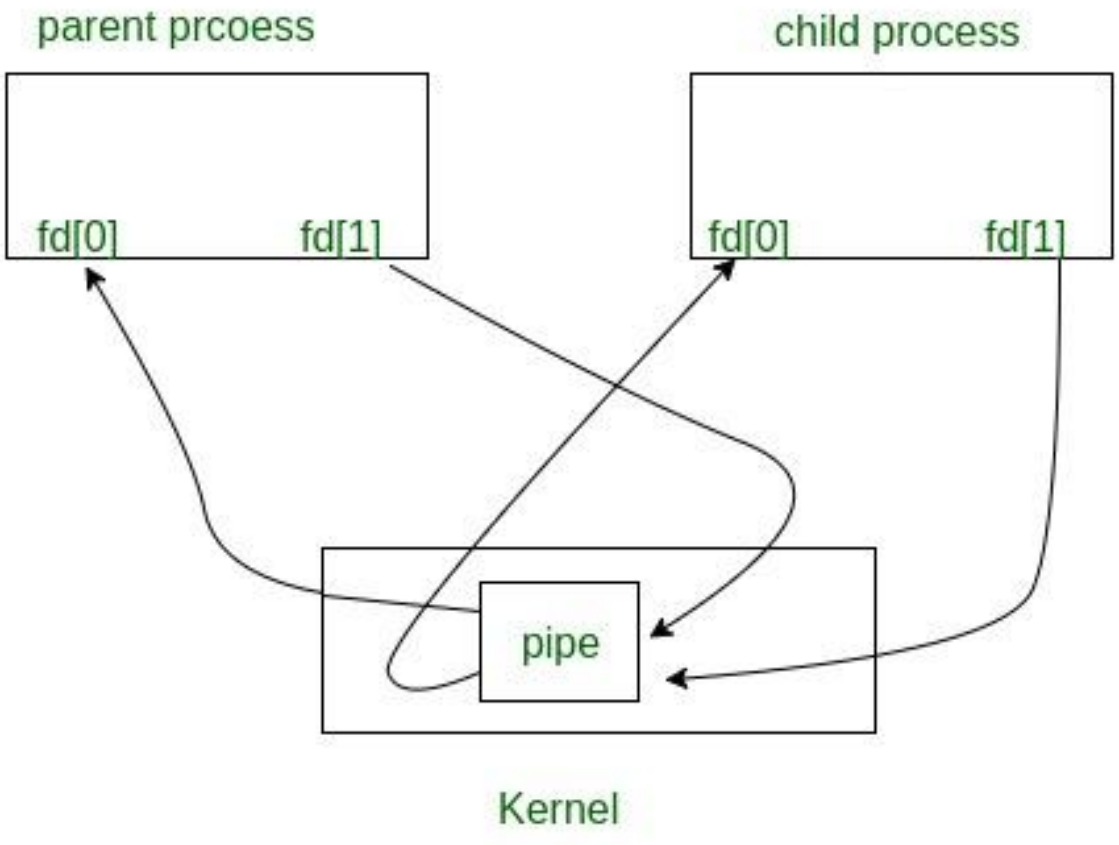
innovate achieve lead

```
#include <stdio.h>
#include <unistd.h>
#define MSGSIZE 16
char* msg1 = "hello, world #1";
char* msg2 = "hello, world #2";
char* msg3 = "hello, world #3";
  int main()
{
    char inbuf[MSGSIZE];
    int p[2], i;

    if (pipe(p) < 0)
        exit(1);
      /* continued */
    /* write pipe */
    write(p[1], msg1, MSGSIZE);
      write(p[1], msg2, MSGSIZE);
      write(p[1], msg3, MSGSIZE);

      for (i = 0; i < 3; i++) {
          /* read pipe */
          read(p[0], inbuf, MSGSIZE);
          printf("% s\n", inbuf);
      }
      return 0;
}
```

Parent and Child sharing a pipe



Example: what is the output of this program?


```
#include <stdio.h>
#include <unistd.h>
#define MSGSIZE 16
char* msg1 = "hello, world #1";
char* msg2 = "hello, world #2";
char* msg3 = "hello, world #3";
  int main()
{
    char inbuf[MSGSIZE];
    int p[2], pid, nbytes;

    if (pipe(p) < 0)
        exit(1);

    /* continued */
    if ((pid = fork()) > 0) {
        write(p[1], msg1, MSGSIZE);
        write(p[1], msg2, MSGSIZE);
        write(p[1], msg3, MSGSIZE);
 // Adding this line will
        // not hang the program
        // close(p[1]);
        wait(NULL);
    }

    else {
        // Adding this line will
        // not hang the program
        // close(p[1]);
        while ((nbytes = read(p[0], inbuf,
MSGSIZE)) > 0)
            printf("% s\n", inbuf);
        if (nbytes != 0)
            exit(2);
        printf("Finished reading\n");
    }
    return 0;
}
```

exit() function in C

innovate achieve lead

1. exit() in C

- The C exit() function is a standard library function used to terminate the calling process.
- When exit() is called, any open file descriptors belonging to the process are closed and any children of the process are inherited by process 1, init, and the process parent is sent a SIGCHLD signal. It is defined inside the <stdlib.h> header file.

Syntax of exit() in C

```
void exit(int status);
```

Parameters

- The exit() function in C only takes a single parameter ***status*** which is the exit code that is returned to the caller i.e. either the operating system or parent process
- There are two valid values we can use as the status each having a different meaning. They are as follows:

0 or EXIT_SUCCESS: 0 or EXIT_SUCCESS means that the program has been successfully executed without encountering any error.

1 or EXIT_FAILURE: 1 or EXIT_FAILURE means that the program has encountered an error and could be executed successfully. // any non-zero value //

1. Flushes unwritten buffered data.
2. Closes all open files.
3. Removes temporary files.
4. Returns an integer exit status to the operating system.

abort() in C

2. abort() in C

- The C abort() function is the standard library function that can be used to exit the C program. But unlike the exit() function, abort() may not close files that are open. It may also not delete temporary files and may not flush the stream buffer. Also, it does not call functions registered with atexit().

Syntax of abort() in C

```
void abort(void);
```

Parameters

- The C abort() function does not take any parameter and does not have any return value. This function actually terminates the process by raising a SIGABRT signal, and your program can include a handler to intercept this signal.
- Return Value

The abort() function in C does not return any value.

assert() in C

3. assert() in C

- The C assert() function is a macro defined inside the <assert.h> header file. It is a function-like macro that is used for debugging. It takes an expression as a parameter,
- If the expression evaluates to 1 (true), the program continue to execute.
- If the expression evaluates to 0 (false), then the expression, source code filename, and line number are sent to the standard error, and then the abort() function is called.

Syntax of assert() in C

void assert(expression);

Parameters

expression: It is the condition we want to evaluate.

Return Value

The assert() function does not return any value.

Blocking pipes

- Blocking pipes are synchronous, which means they execute in order
- For example, if two processes at opposite ends of a pipe are generating and consuming data at different rates, the faster process will automatically wait for the slower process to catch up
- This can simplify congestion control and prevent data loss
- However, blocking pipes can be slow because reading from a pipe is not predictable
- For example, if the pipe is empty, the kernel doesn't know when more data will be added

Non-blocking pipes

- Non-blocking pipes are asynchronous, which means they execute without waiting on others and do not process results immediately
- For example, in non-blocking I/O, a system call that would normally block while waiting for I/O instead returns immediately with an error code