Inter-process communication

- Wait for exit status (report when done)
- Pipe (always open for communication)
- Signals (send when you want, handle or ignore)
- Sockets (open connection with the world)

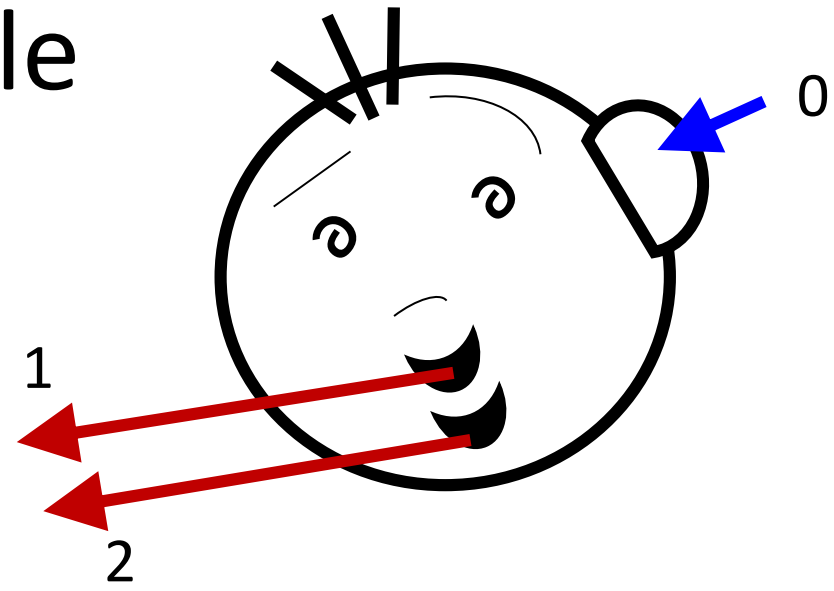
File descriptors. Pipes

Lecture 04.03

File descriptor table

#	Data Stream
0	The keyboard
1	The screen
2	The screen
3	Database connection

The process might also have other open streams



Redirection just replaces data streams in file descriptor table

- ./my_prog < stories.txt >out.txt 2>log.txt

#	Data Stream
0	~~The keyboard~~ File stories.txt
1	~~The screen~~ File out.txt
2	~~The screen~~ File log.txt

fileno() tells you the descriptor of an open file

- Every time you open a file, the operating system registers a new item in the descriptor table:

```
FILE *my_file = fopen("guitar.mp3", "r");
int descriptor = fileno(my_file);
```

This will return 4

#	Data Stream
0	The keyboard
1	The screen
2	The screen
3	Database connection
4	File guitar.mp3

We can do redirection inside the same process by rewriting descriptor table

dup2(4, 3); //returns -1 on failure

#	Data Stream
0	The keyboard
1	The screen
2	The screen
3	File guitar.mp3
4	File guitar.mp3

Silently closes Database connection

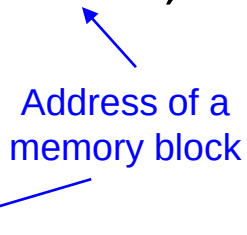
Example: redirect standard output to a file

```
FILE *f = fopen("stories.txt", "w");
dup2 (fileno(f), 1);
```

Using file descriptors

```
int output_fd = open (f_name, O_WRONLY|O_CREAT);
write (output_fd, buffer, buffer_size_bytes);

int input_fd = open (f_name, O_RDONLY);
read (input_fd, buffer, buffer_size_bytes);
```




```
write (1, buffer, buffer_size_bytes);
read (0, buffer, buffer_size_bytes);
```

pipe() opens two data streams

- The pipe(fds) functions creates two connected streams and adds them to the table
- Whatever is written into one stream can be read from the other

#	Data Stream
0	The keyboard
1	The screen
2	The screen
3	Read end of a pipe
4	Write-end of a pipe

Recording file descriptors

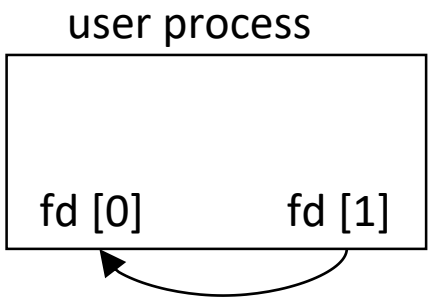
- When pipe() creates the two lines in the descriptor table, it will store their file descriptors in a two-element array:

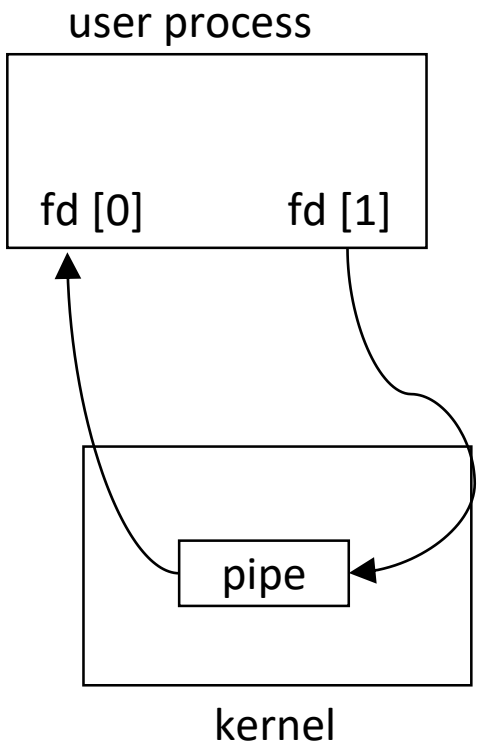
```
int fd[2];
if (pipe(fd) == -1) {
      error("Can't create the pipe");
}
```

fd[1] = 4 writes to the pipe
fd[0] = 3 reads from it

#	Data Stream
0	The keyboard
1	The screen
2	The screen
3	Read end of a pipe
4	Write-end of a pipe

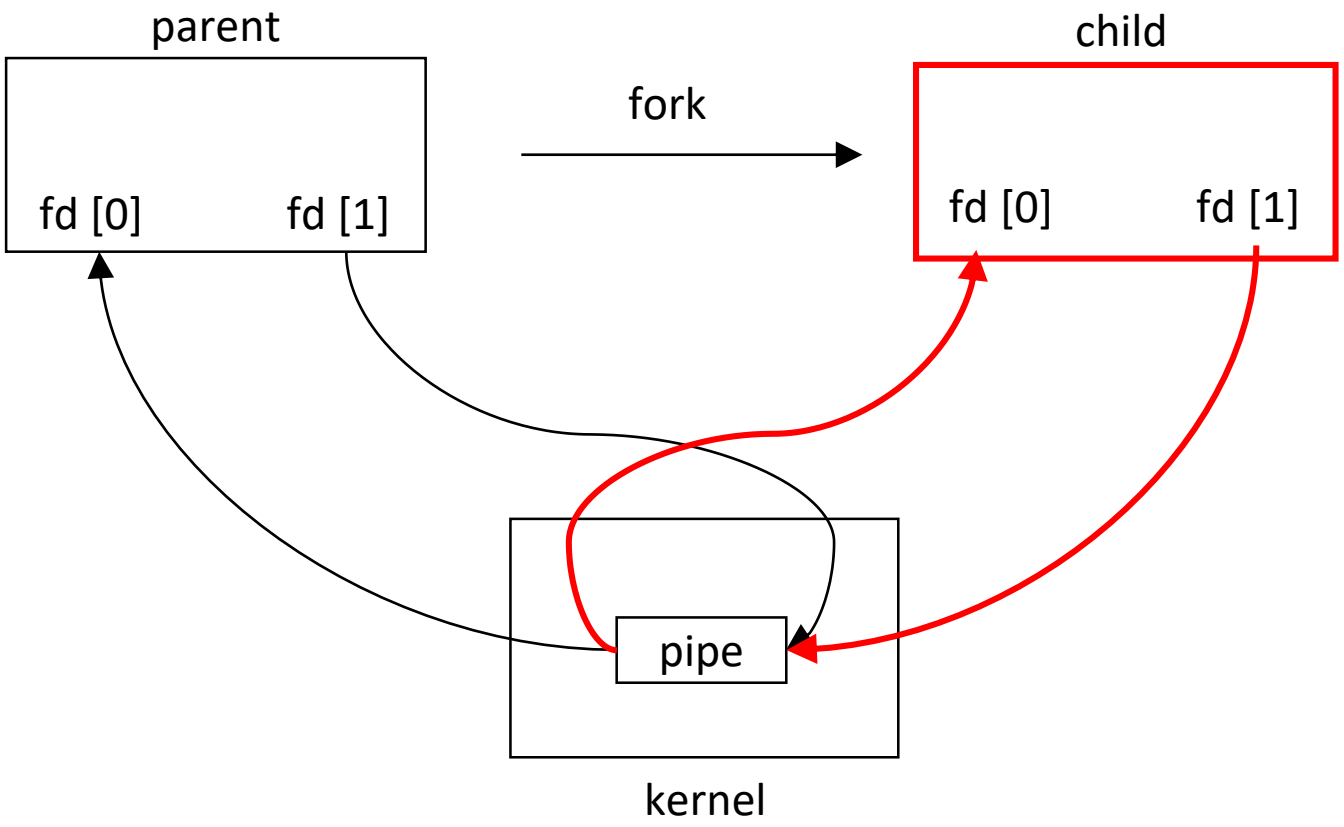
Pipe is a buffer in a kernel space



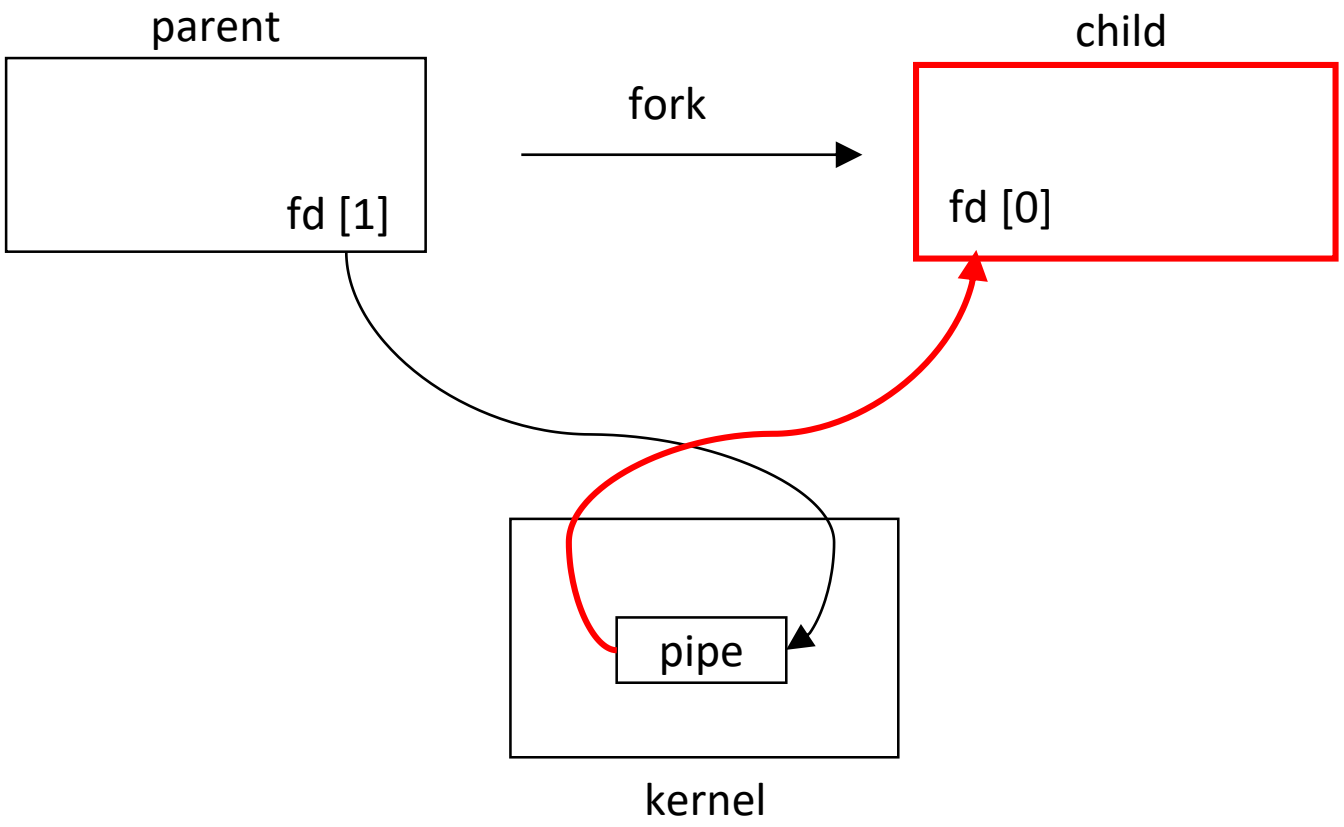


- Every read from a pipe copies from kernel space to user space
- Every write to a pipe copies from user space to kernel space

Pipe and fork



Pipe and fork



Half-duplex communication

- Pipes become useful by exploiting the fact that file descriptors are inherited through fork.

1. *pipe*(fds)
2. *fork*()
3. parent: *close*(fds[0])
4. child: *close*(fds[1])
5. parent can transfer data to child with *write*(fds[1], ...)
6. child can receive data from parent with *read*(fds[0], ...)

(exchange 0 and 1 for child to parent data transfer)

Piped commands are parents and children

ls | wc -l

- Whenever you pipe commands together on the command line, you are actually connecting them together as parent and child processes
- In the above example, the wc -l command is the parent of the ls

1. The shell creates the parent process
2. The parent process forks the ls in a child process
3. The parent connects the output of the child with the input of the parent using a pipe
4. The child process execs the ls
5. The parent process execs the wc -l command

Example: ls | wc -l

```
int pfds[2];
 pipe(pfds);
 if (!fork()) {
    dup2(pfds[1],1);         /* make stdout same as pfds[1] */
    close(pfds[0]);                    /* we don't need this */
    execlp ("ls", "ls", NULL);
 } else {
    dup2(pfds[0],0);                   /* make stdin same as pfds[0] */
    close(pfds[1]);                    /* we don't need this */
    execlp ("wc", "wc", "-l", NULL);
 }
```

child

parent

Pipes: summary

- The byte stream written to one end of the pipe can be read from the other
- Once created, pipes are referenced by file descriptor handles
- Pipes are accessible only by related processes
- No identifier is used to rendez-vous on pipes, they are requested directly to the kernel
- Pipes are process-persistent: they disappear when related processes terminate

What does this program do?

```
int main( void ) {
        int n, fd [ 2 ] ;
        pid_t pid ;
        char line [MAXLINE ] ;
        pid = fork ( );
        pipe ( fd ) ;
        i f ( pid > 0) { /* parent */
                close ( fd [ 0 ] ) ;
                write ( fd [1] , " Hello , World ! \n" , 14) ;
        } else { /* child */
                close ( fd [ 1 ] ) ;
                n = read ( fd [0] , line , MAXLINE ) ;
                write (STDOUT_FILENO, line , n ) ;
        }
        exit ( EXIT_SUCCESS ) ;   }
```

Is there something wrong with it?

```
int main( void ) {
        int n, fd [ 2 ] ;
        pid_t pid ;
        char line [MAXLINE ] ;
        pid = fork ( );
        pipe ( fd ) ;
        i f ( pid > 0) { /* parent */
                close ( fd [ 0 ] ) ;
                write ( fd [1] , " Hello , World ! \n" , 14) ;
        } else { /* child */
                close ( fd [ 1 ] ) ;
                n = read ( fd [0] , line , MAXLINE ) ;
                write (STDOUT_FILENO, line , n ) ;
        }
        exit ( EXIT_SUCCESS ) ;   }
```

Full-duplex communication with pipes

- Once more: pipes are *half-duplex*: one pipe can be used to transfer data **in one direction only** - either from parent to child or from child to parent
- To do full-duplex communication with pipes (i.e. transfer data in both directions), **2 pipe calls before** fork are needed

Full-duplex pipe recipe

1. pipe(p2c)
2. pipe(c2p)
3. fork()
4. parent: close(p2c[0]); close(c2p[1])
5. child: close(p2c[1]); close(c2p[0])
6. parent ! child: write(p2c[1], ...)
7. child ! parent: write(c2p[1], ...)

Exercise

- We wrote a program that forked one child for each command line argument. The child computes the length of the command line argument and exits with that integer as the return value. The parent sums these return codes and reports the total length of all the command line arguments together.
- Now, we will do the same program except that each child will communicate the length to the parent through a pipe.

Define 2D array of ints to store argc file descriptors

```
int fds[argc][2];
```

We will use this array starting from index 1 - for simplicity

Loop through command-line arguments starting from 1

```
for (int i=1; i<argc; i++) {
        //create pipe i
        pipe (fds[i]);

        //fork a new child process
        int result = fork();
```

In a child process at iteration *i*

```
//close reading end of pipe i
close (fds[i][0]);

//work on the i-th argument
int len = strlen (argv[i]);

//write result to the pipe
write(fds[i][1], &len, sizeof(int));
```

Cleaner version of a child

```
//close reading end of pipe i
close (fds[i][0]);

// Before we forked, parent had opened the reading ends to
 all previously forked children; so close those.
        for (int j=1; j< i; j++)
                close (fds[j][0]);
```

Exit child process to avoid fork in the next iteration

```
close(fds[i][0]);

for (int j=1; j< i; j++)
    close (fds[j][0]);

int len = strlen(argv[i]);

write(fds[i][1], &len, sizeof(int));
close(fds[i][1]);
exit(0);
```

In the parent process

```
// close the end of the pipe that we don't need
close(fds[i][1]);
```

After the for loop: parent reads from all pipe buffers

```
int res;
 for (int i = 1; i < argc ; i++) {
        read(fds[i][0], &res, sizeof(int));
        sum += res;
 }
```

Inter-process communication

- Wait for exit status (report when done)
- Pipe (always open for communication)
- Signals (send when you want, handle or ignore)
- Sockets (open connection with the world)