CS 332/532 Systems Programming

Lecture 28

I/O Redirection / 2

Professor : Mahmut Unan – UAB CS

Agenda

- Sharing between parent and child processes
- Dup2
- Pipes

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

Exercise

- This example illustrate how to use dup2 to replace the stdin and stdout file descriptors with the files stdin.txt and stdout.txt, respectively.
- We use the file forkexecvp.c from the previous lab and the new lines of code are highlighted.

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

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

    /* display program usage if arguments are missing */
    if (argc < 2) {
        printf("Usage: %s <command> [args]\n", argv[0]);
        exit(-1);
    }

    /* open file to read standard input stream,
       make sure the file stdin.txt exists, even if it is empty */
    if ((fdin = open("stdin.txt", O_RDONLY)) == -1) {
       printf("Error opening file stdin.txt for input\n");
       exit(-1);
    }
```

```
26
27      /* open file to write standard output stream in append mode.
28         create a new file if the file does not exist. */
29      if ((fdout = open("stdout.txt", O_CREAT | O_APPEND | O_WRONLY, 0755)) == -1) {
30         printf("Error opening file stdout.txt for output\n");
31         exit(-1);
32      }
33
34      pid = fork();
35      if (pid == 0) { /* this is child process */
36          /* replace standard input stream with the file stdin.txt */
37          dup2(fdin, 0);
38
39          /* replace standard output stream with the file stdout.txt */
40          dup2(fdout, 1);
41
42          execvp(argv[1], &argv[1]);
43          /* since stdout is written to stdout.txt and not the terminal,
44             we should write to stderr in case exec fails, we use perror
45             that writes the error message to stderr */
46      perror("exec");
47      exit(-1);
```

```
48      } else if (pid > 0) { /* this is the parent process */
49          /* output from the parent process still goes to stdout :-) */
50          printf("Wait for the child process to terminate\n");
51          wait(&status); /* wait for the child process to terminate */
52          if (WIFEXITED(status)) { /* child process terminated normally */
53              printf("Child process exited with status = %d\n", WEXITSTATUS(status));
54              /* parent process still has the file handle to stdout.txt,
55                 now that the child process is done, let us write to
56                 the file stdout.txt using the write system call */
57              write(fdout, "Hey! This is the parent process\n", 32);
58              close(fdout);
59              /* since we opened the file in append mode, the above text
60                 will be added after the output from the child process */
61          } else { /* child process did not terminate normally */
62              printf("Child process did not terminate normally!\n");
63              /* look at the man page for wait (man 2 wait) to determine
64                 how the child process was terminated */
65          }
66      } else { /* we have an error */
67          perror("fork"); /* use perror to print the system error message */
68          exit(EXIT_FAILURE);
69      }
70
71      return 0;
72  }
```

compile & run

- Compile and run this program using myprog as the child process. Note that you have provide input to *myprog* in the file *stdin.txt* and the output of *myprog* will be written to *stdout.txt*. As we did not do anything with the *stderr* stream, the output to *stderr* stream goes to the terminal. You can add the PID in the print statement to confirm which output is sent to the terminal and which output is sent to the files. Here is a terminal session that illustrates this interaction:

recall myprog.c

```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 & run

```
(base) mahmutunan@MacBook-Pro lecture27 % gcc -Wall -o myprog myprog.c
(base) mahmutunan@MacBook-Pro lecture27 % gcc -Wall -o ioredirect ioredirect.c
(base) mahmutunan@MacBook-Pro lecture27 % cat > stdin.txt
Mahmut
(base) mahmutunan@MacBook-Pro lecture27 % ./ioredirect ./myprog
Wait for the child process to terminate
[stderr]: Hello Mahmut!
Child process exited with status = 0
(base) mahmutunan@MacBook-Pro lecture27 % cat stdout.txt
Please enter your name: [stdout]: Hello Mahmut!
Hey! This is the parent process
```

```
(base) mahmutunan@MacBook-Pro lecture27 % cat > stdin.txt
World
(base) mahmutunan@MacBook-Pro lecture27 % ./ioredirect ./myprog
Wait for the child process to terminate
[stderr]: Hello World!
Child process exited with status = 0
(base) mahmutunan@MacBook-Pro lecture27 % cat stdout.txt
Please enter your name: [stdout]: Hello Mahmut!
Hey! This is the parent process
Please enter your name: [stdout]: Hello World!
Hey! This is the parent process
(base) mahmutunan@MacBook-Pro lecture27 % _
```

```
(base) mahmutunan@MacBook-Pro lecture27 % ./ioredirect uname -a
Wait for the child process to terminate
Child process exited with status = 0
(base) mahmutunan@MacBook-Pro lecture27 % cat stdout.txt
Please enter your name: [stdout]: Hello Mahmut!
Hey! This is the parent process
Please enter your name: [stdout]: Hello World!
Hey! This is the parent process
Darwin MacBook-Pro.local 19.6.0 Darwin Kernel Version 19.6.0: Mon Aug 31 22:12:52 PDT 2020; roo
t:xnu-6153.141.2~1/RELEASE_X86_64 x86_64
Hey! This is the parent process
(base) mahmutunan@MacBook-Pro lecture27 % _
```

Pipes

- We have seen the Linux shell support pipes.
- For example:

```
$ ps -elf | grep ssh
```

- The above example redirects the output of the program ps to another program grep (instead of sending the output to standard output).
- Similarly, the program grep uses the output of ps as the input instead of a file name as the argument. The shell implements this redirection using pipes.
- The system call pipe is used to create a pipe and in most Linux systems pipes provide a unidirectional flow of data between two processes.

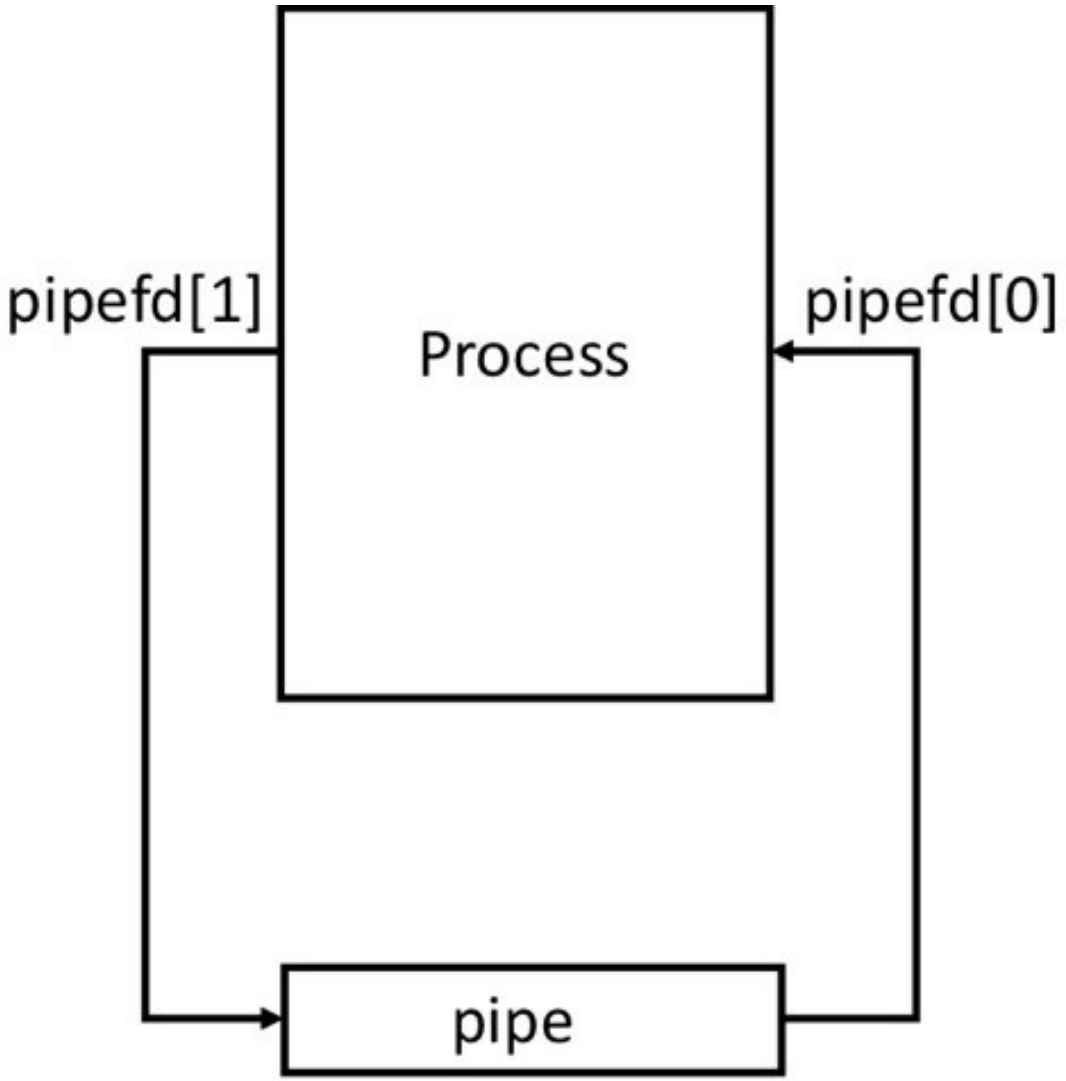
The C API for the pipe function

- The C API for the pipe function is shown below:

```
#include <unistd.h>
int pipe(int pipefd[2]);
```

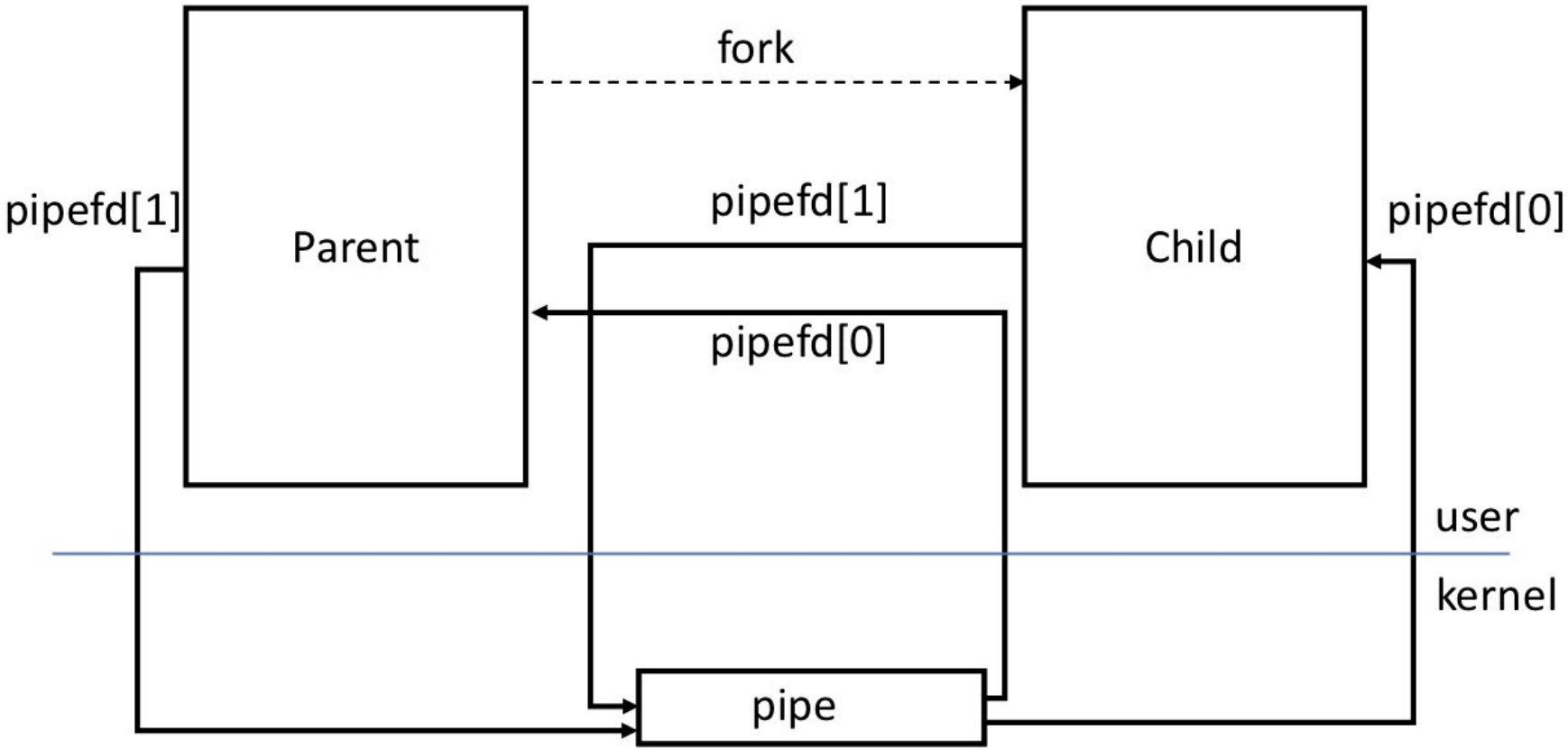
- The pipe call returns two file descriptors corresponding to the read and write ends of the pipe.
- The first file descriptor (*pipefd[0]*) refers to the read end of the pipe (can be used for reading data from the pipe) and the second file descriptor (*pipefd[1]*) refers to the write end of the pipe (can be used for writing data to the pipe).
- The kernel buffers the data written to the pipe until it is read from the read end of the pipe. When there is an error in creating the pipe, it returns -1 and sets the corresponding *errno*, otherwise it returns 0 on success.

- The diagram below illustrates the creation of a pipe in a single process.

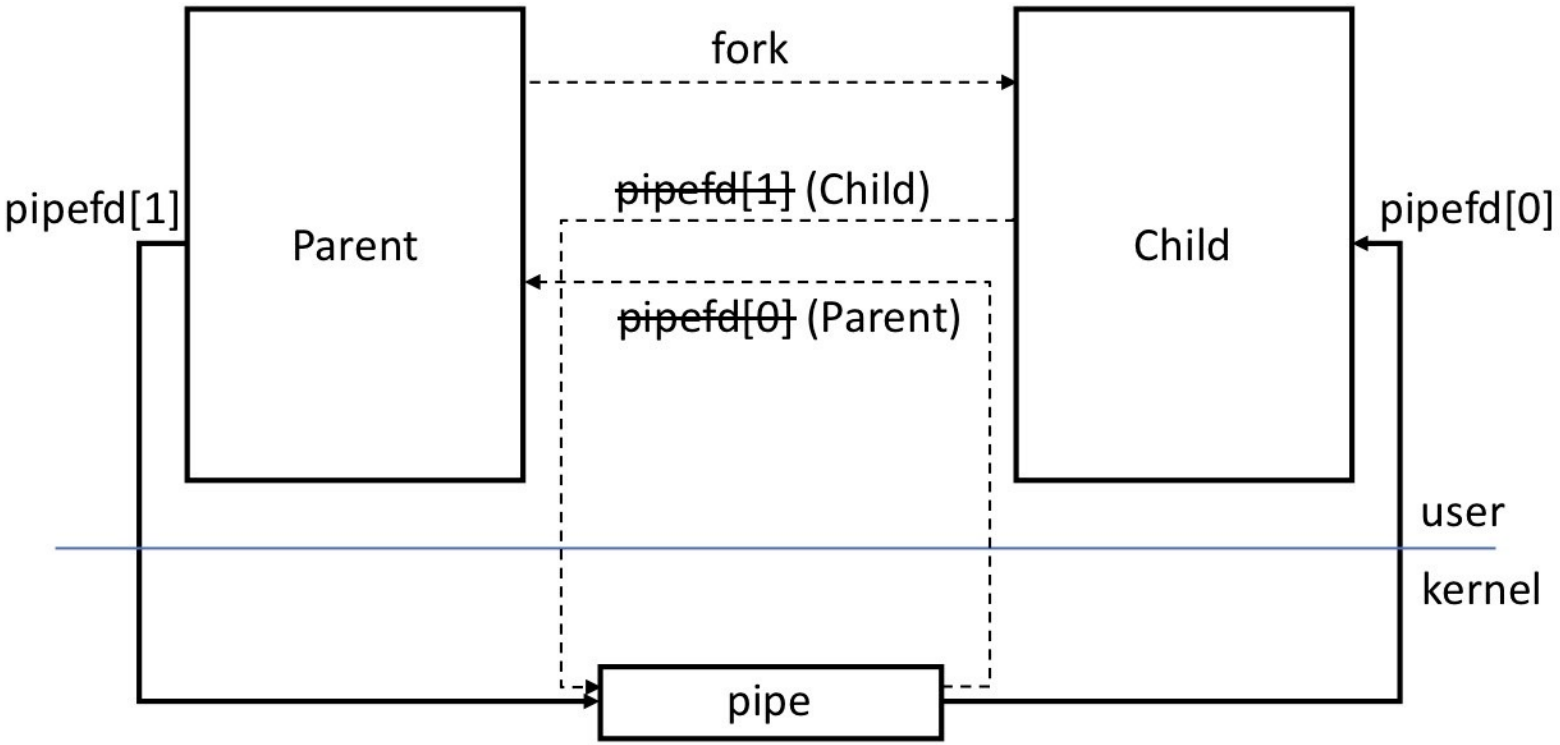


- We really don't need to create a pipe to communicate within the same process, typically we use pipes to communicate between a parent process and a child process.
- In such a case, first a pipe is created by the parent process and then it creates a child process using the fork command.
- Since fork creates a copy of the parent process, the child process will also inherit the all open file descriptors and will have access to the pipe

parent - child



- To provide a unidirectional data channel for communication between the two process, the parent process closes the read end of the pipe and the child process closes the write end of the pipe as shown in the diagram below.



Exercise 1

- The following example shows the steps involved in creating a pipe, forking a child process, closing the file descriptors in the parent and child process, and communication between the parent and child process.
- The parent process writes the string passed as the command-line argument to the pipe and the child process reads the string from the pipe, converts the string to uppercase, and prints it to the standard output.
- The read and write functions that operate on files are used to read and write data from the pipe.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <ctype.h>
#include <sys/wait.h>
#include <sys/stat.h>

int main(int argc, char **argv) {
  pid_t pid;
  int status;
  int pipefd[2]; /* pipefd[0] for read, pipefd[1] for write */
  char c;

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

  if (pipe(pipefd) == 0) { /* Open a pipe */
    if ((pid = fork()) == 0) { /* I am the child process */
      close(pipefd[1]);    /* close write end */

```

```
23
24        while (read(pipefd[0], &c, 1) > 0) {
25            c = toupper(c);
26            write(1, &c, 1);
27        }
28        write(1, "\n", 1);
29        close(pipefd[0]);
30
31        exit(EXIT_SUCCESS);
32    } else if (pid > 0) { /* I am the parent process */
33        close(pipefd[0]);   /* close read end */
34
35        write(pipefd[1], argv[1], strlen(argv[1]));
36        close(pipefd[1]);
37
38        wait(&status);      /* wait for child to terminate */
39        if (WIFEXITED(status))
40            printf("Child process exited with status = %d\n", WEXITSTATUS(status));
```

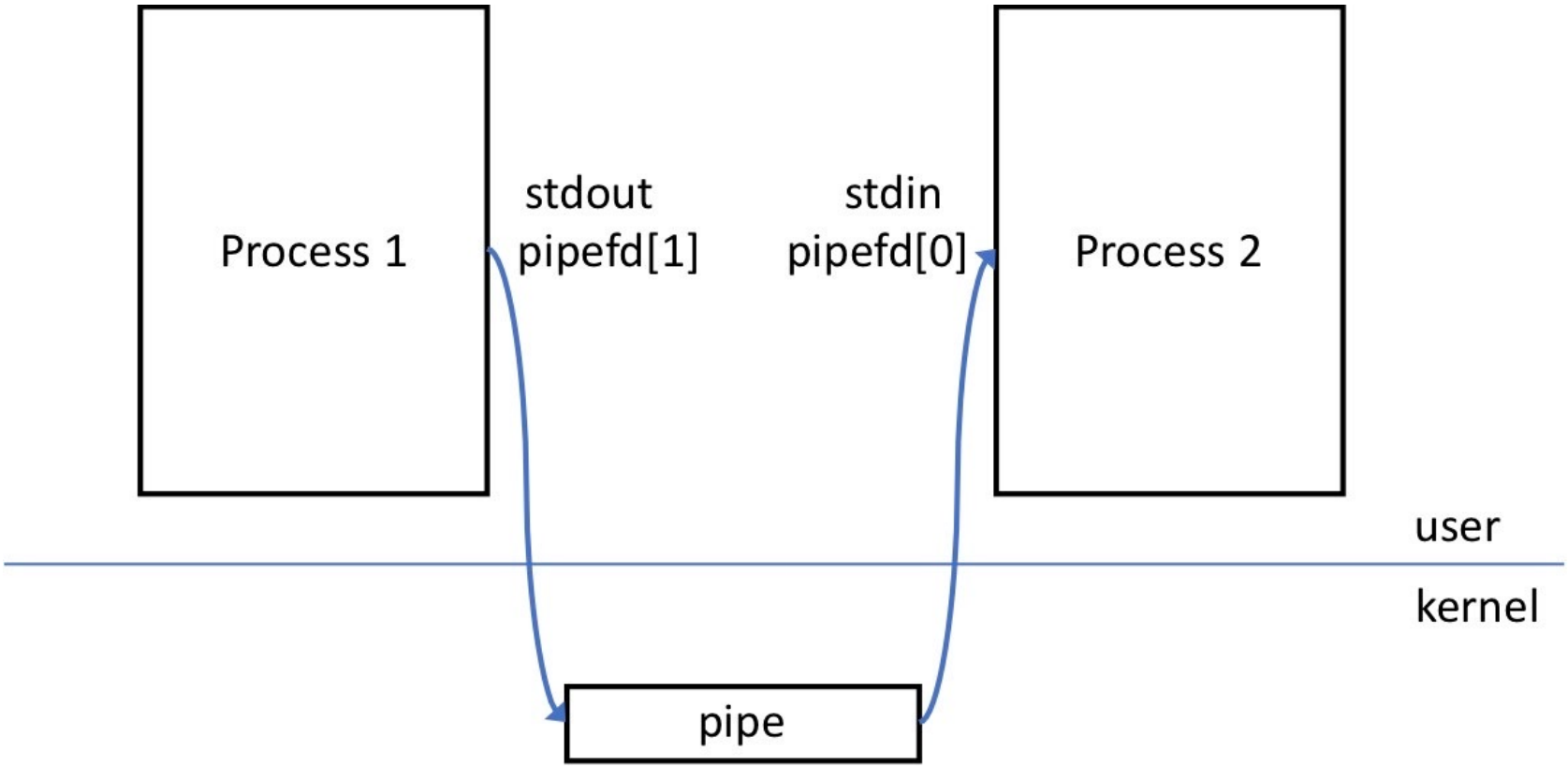
```
41          else
42              printf("Child process did not terminate normally!\n");
43        } else { /* we have an error in fork */
44          perror("fork");
45          exit(EXIT_FAILURE);
46        }
47      } else {
48        perror("pipe");
49        exit(EXIT_FAILURE);
50      }
51
52      exit(EXIT_SUCCESS);
53    }
54
```

```
(base) mahmutunan@MacBook-Pro lecture28 % gcc -Wall pipe1.c -o pipe1

(base) mahmutunan@MacBook-Pro lecture28 % ./pipe1 cs332lowercase
CS332LOWERCASE
Child process exited with status = 0
(base) mahmutunan@MacBook-Pro lecture28 %
```

- The above example shows how the pipe is used by a parent and a child process to communicate.
- We can further extend this to implement pipes between any two programs such that the output of one program is redirected to the input of another program (e.g., ps -elf | grep ssh).
- In order to do this, we have to replace the standard output of the first program with the write end of the pipe and replace the standard input of the second program with the read end of the pipe.
- We have seen in the previous lecture/lab that this can be done using the dup2 system call.
- We will use the dup2 to perform this redirection and implement the pipe operation between two processes

- The pipe operation between two processes as shown in the diagram below.



- We have several options to create the two processes, some of the possible options include:

1. The parent process creates a child process, the child process uses exec to launch the second program, and the parent process will use exec to launch the first program.
2. The parent process creates two child process, the first child process uses exec to launch the first program, the second child process uses exec to launch the second program, and the parent process waits for the two child processes to terminate.
3. The parent process create a child process, the child process creates another child process which in turn uses exec to launch the first program, then the child process uses exec to launch the second program, and the parent waits for the child process to terminate.

In this example, we will use the first option

Exercise 2

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

int main(int argc, char **argv) {
    pid_t pid;
    int pipefd[2]; /* fildes[0] for read, fildes[1] for write */

    if (argc != 3) {
        printf("Usage: %s <command1> <command2>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    if (pipe(pipefd) == 0) { /* Open a pipe */
      pid = fork(); /* fork child process to execute command2 */
      if (pid == 0) { /* this is the child process */
        /* close write end of the pipe */
    close(pipefd[1]);

        /* replace stdin with read end of pipe */
    if (dup2(pipefd[0], 0) == -1) {
           perror("dup2");
           exit(EXIT_FAILURE);
        }
```

Exercise 2

```
28
29          /* execute <command2> */
30          execlp(argv[2], argv[2], (char *)NULL);
31          perror("execlp");
32          exit(EXIT_FAILURE);
33        } else if (pid > 0) { /* this is the parent process */
34          /* close read end of the pipe */
35      close(pipefd[0]);
36
37      /* replace stdout with write end of pipe */
38      if (dup2(pipefd[1], 1) == -1) {
39              perror("dup2");
40              exit(EXIT_FAILURE);
41          }
42
43          /* execute <command1> */
44          execlp(argv[1], argv[1], (char *)NULL);
45          perror("execlp");
46          exit(EXIT_FAILURE);
47        } else if (pid < 0) { /* we have an error */
48          perror("fork"); /* use perror to print the system error message */
49          exit(EXIT_FAILURE);
50        }
51      } else {
52        perror("pipe");
53        exit(EXIT_FAILURE);
54      }
55
56      return 0;
57  }
```

compile & run

pipe1.c | pipe0.c | p2.c | p1.c

```c
#include <stdio.h>

int main(int argc, char **argv) {
    int a = 15, b = 25;
    printf("%d\n", a+b);

}
```

pipe1.c | pipe0.c | p2.c | p1.c

```c
#include <stdio.h>

int main(int argc, char **argv) {
    int x;
    printf("Enter an int \n");
    scanf("%d",&x);
    printf("square of your number is %d \n", x*x);

}
```

```
(base) mahmutunan@MacBook-Pro lecture28 % gcc p1.c -o p1
(base) mahmutunan@MacBook-Pro lecture28 % ./p1
40
(base) mahmutunan@MacBook-Pro lecture28 % gcc p2.c -o p2
(base) mahmutunan@MacBook-Pro lecture28 % ./p2
Enter an int
5
square of your number is 25
(base) mahmutunan@MacBook-Pro lecture28 % gcc pipe0.c -o pipe0
(base) mahmutunan@MacBook-Pro lecture28 % ./pipe0 ./p1 ./p2
Enter an int
square of your number is 1600
(base) mahmutunan@MacBook-Pro lecture28 %
```