CS 332/532 Systems Programming

Lecture 29

Pipes / 2

Professor : Mahmut Unan – UAB CS

Agenda

- Pipes continued
- popen
- Pclose

Thread

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.

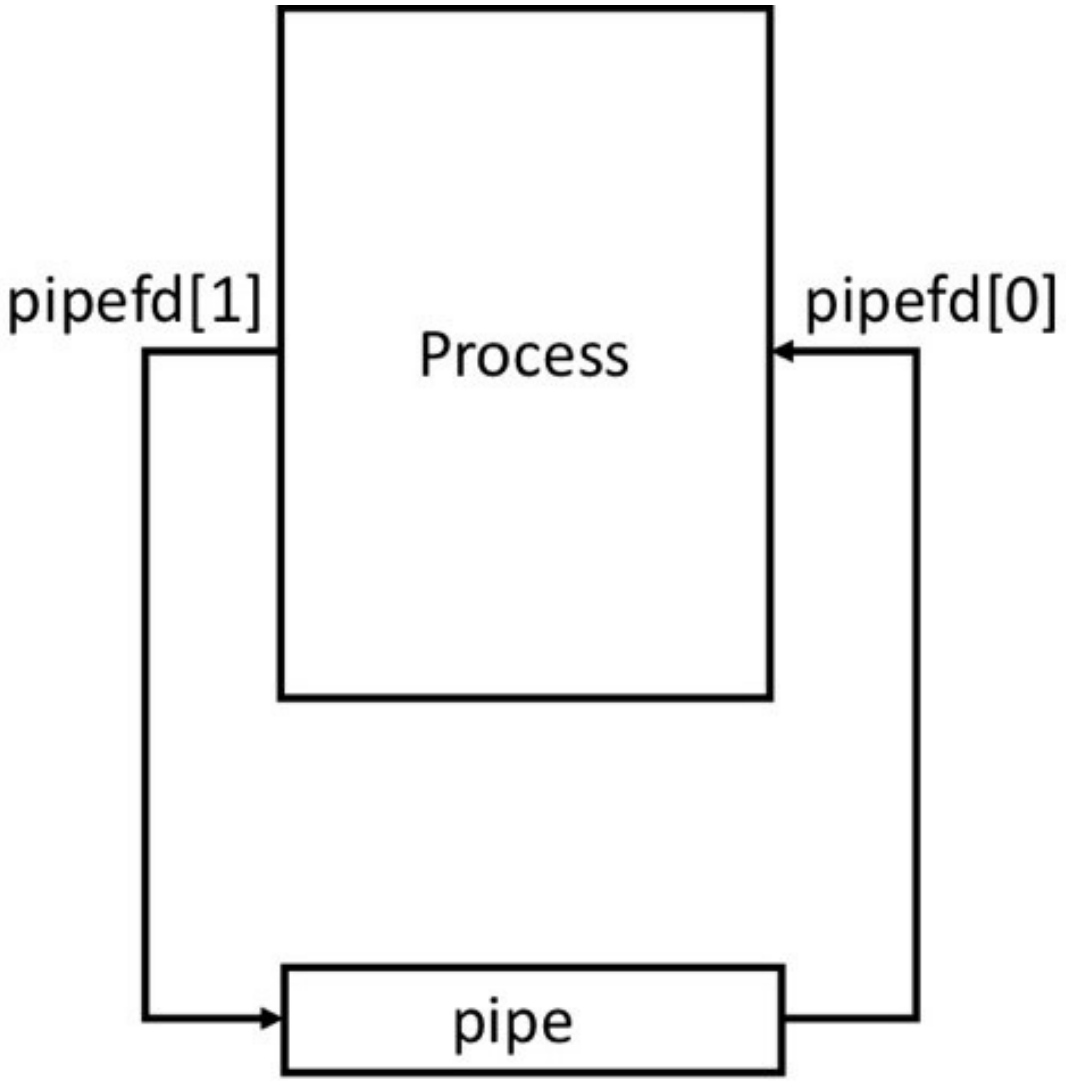
The C API for the pipe function

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

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

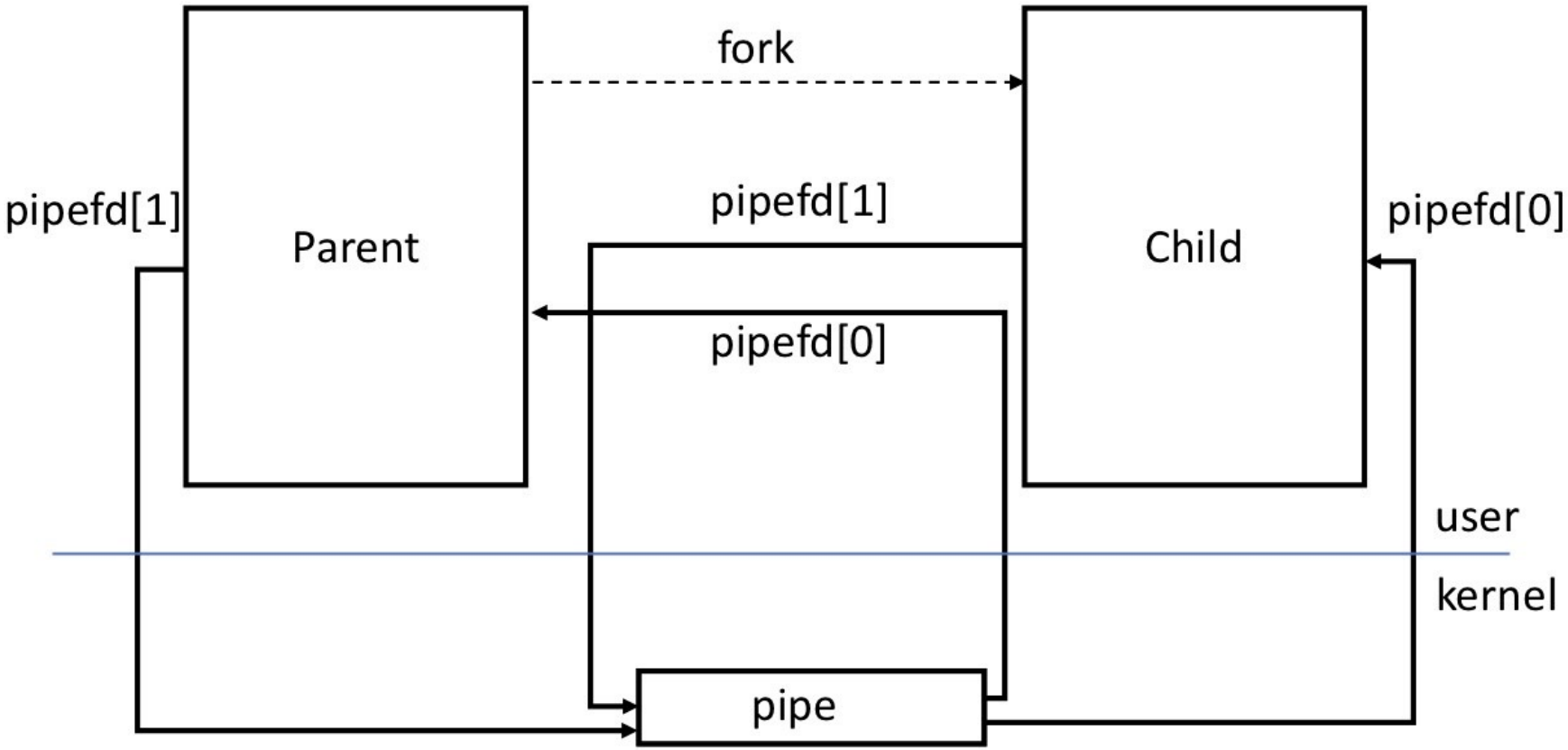
- The pipe call returns two file descriptors corresponding to the read and write ends of the pipe.
- The first file descriptor (*pipefd[0]*) refers to the read end of the pipe (can be used for reading data from the pipe) and the second file descriptor (*pipefd[1]*) refers to the write end of the pipe (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.

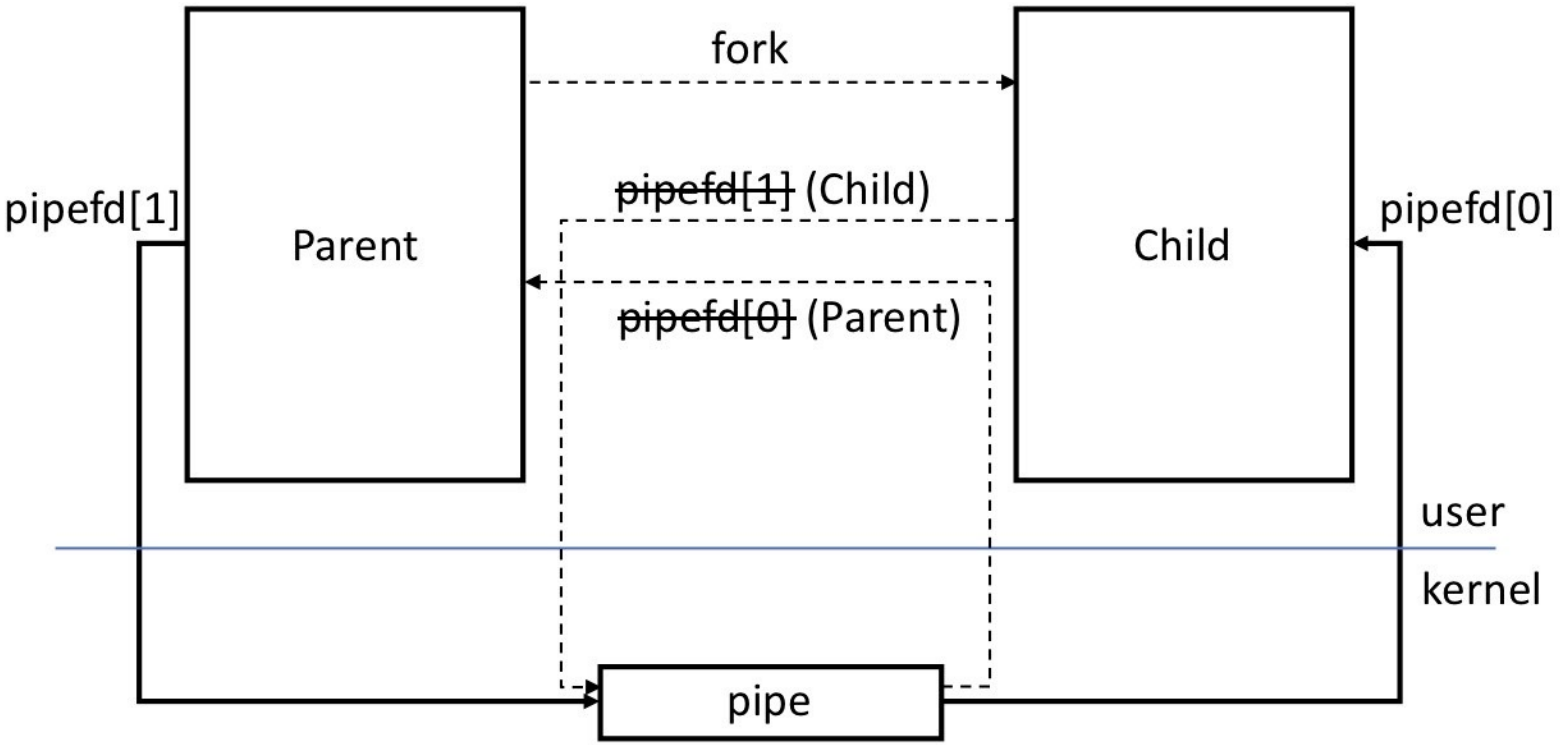


- We really don't need to create a pipe to communicate within the same process, typically we use pipes to communicate between a parent process and a child process.
- In such a case, first a pipe is created by the parent process and then it creates a child process using the fork command.
- Since fork 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 */

```

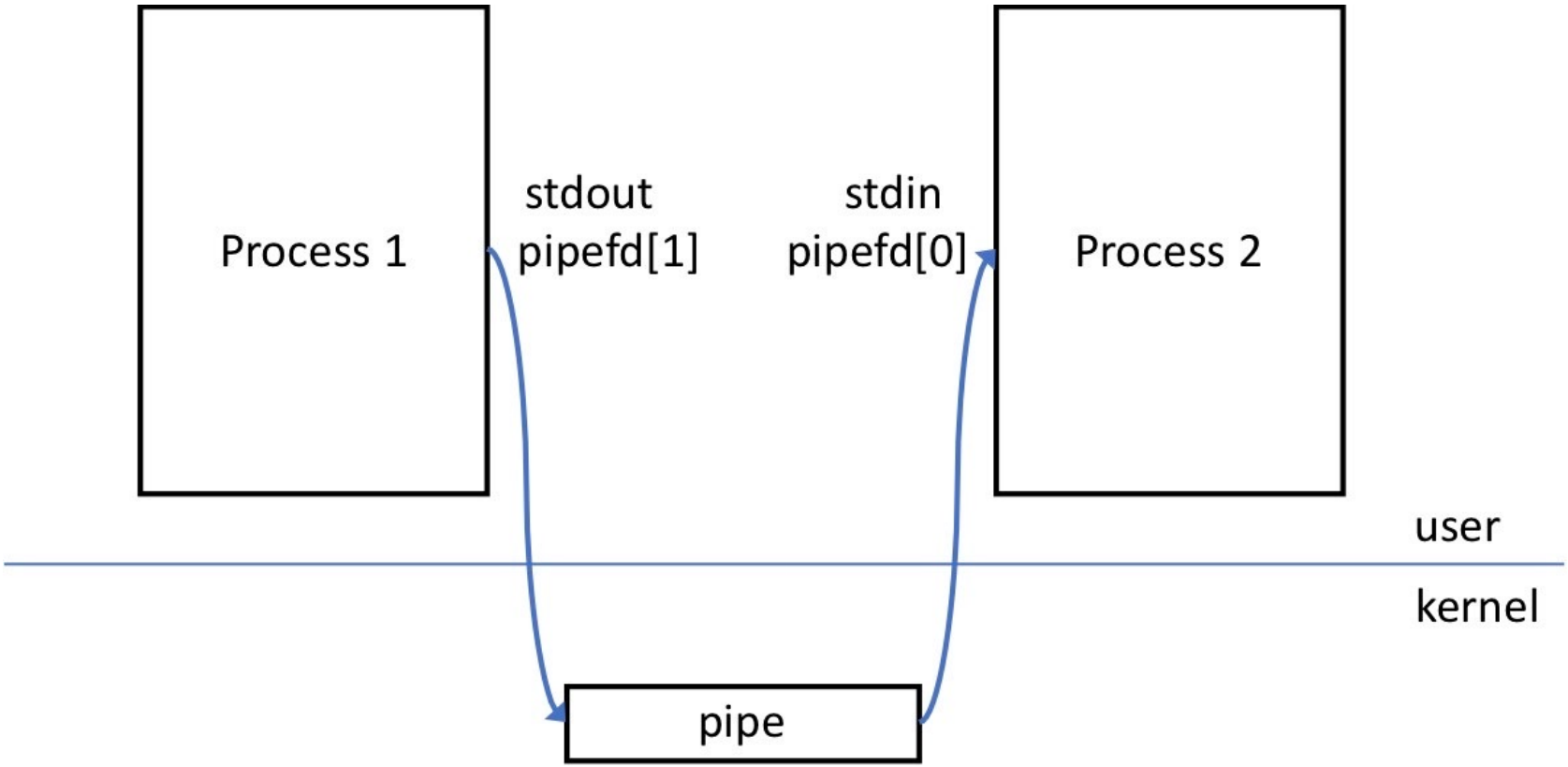
```
23
24      while (read(pipefd[0], &c, 1) > 0) {
25          c = toupper(c);
26          write(1, &c, 1);
27      }
28      write(1, "\n", 1);
29      close(pipefd[0]);
30
31      exit(EXIT_SUCCESS);
32    } else if (pid > 0) { /* I am the parent process */
33      close(pipefd[0]);   /* close read 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

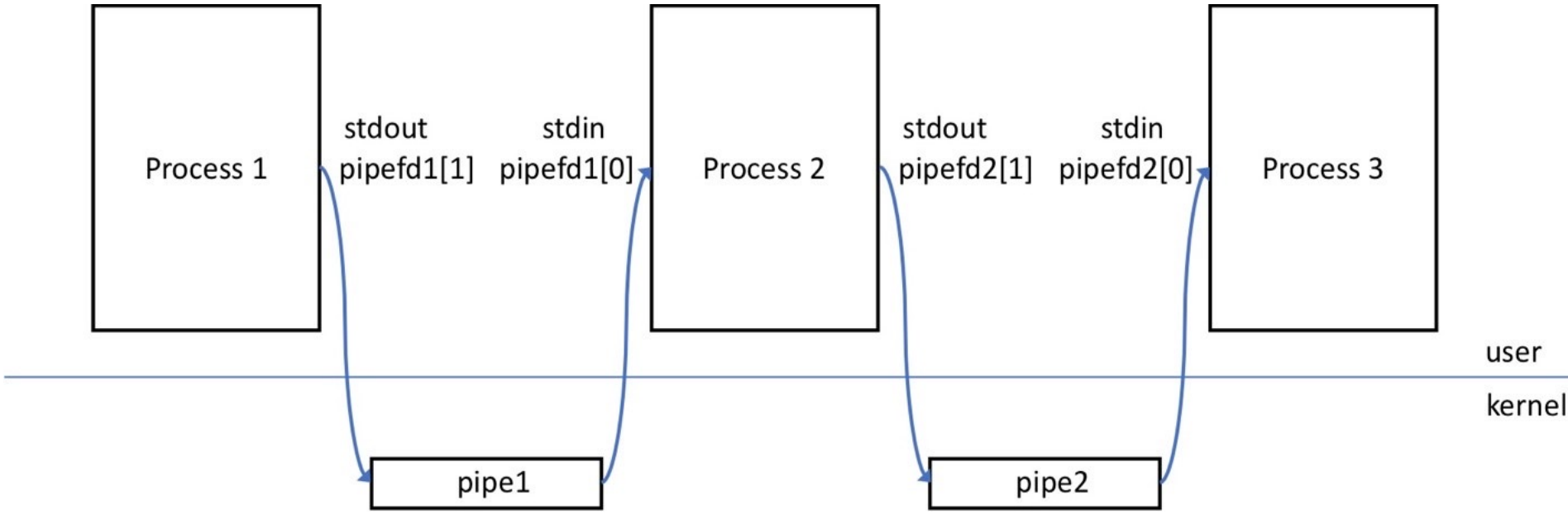
```c
#include <stdio.h>

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 %
```

extend to three process

- We can extend this to three processes if we like to implement something like: *ls | sort | wc.*
- We will create three processes and use two pipes - one for communication between the first and second process and one for communication between second and third process.
- The code for the first and third children will be similar to the example above while the second child has to replace both standard input and standard output streams instead of just one. T

The diagram below illustrates this



pipe3.c

```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 pid1, pid2, pid3;
    int pipefd1[2]; /* pipefd1[0] for read, pipefd1[1] for write */
    int pipefd2[2]; /* pipefd2[0] for read, pipefd2[1] for write */
    int status1, status2, status3;

    if (argc != 4) {
        printf("Usage: %s <command1> <command2> <command3>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    if (pipe(pipefd1) != 0) { /* Open pipefd1 */
      perror("pipe");
      exit(EXIT_FAILURE);
    }

    if (pipe(pipefd2) != 0) { /* Open pipefd2 */
      perror("pipe");
      exit(EXIT_FAILURE);
    }

    pid1 = fork(); /* fork first process to execute command1 */
    if (pid1 == 0) { /* this is the child process */
      /* close read end of the pipefd1 */
      close(pipefd1[0]);

      /* close both ends of pipefd2 */
      close(pipefd2[0]);
      close(pipefd2[1]);
```

```
39          /* replace stdout with write end of pipefd1 */
40          if (dup2(pipefd1[1], 1) == -1) {
41      perror("dup2");
42      exit(EXIT_FAILURE);
43          }
44
45          /* execute <command1> */
46          execlp( file: argv[1],  arg0: argv[1], (char *)NULL);
47          perror("execlp");
48          exit(EXIT_FAILURE);
49
50      } else if (pid1 < 0) { /* we have an error */
51          perror("fork"); /* use perror to print the system error message */
52          exit(EXIT_FAILURE);
53      }
54
55      pid2 = fork(); /* fork second process to execute command2 */
56      if (pid2 == 0) { /* this is child process */
57          /* close write end of the pipefd1 */
58          close(pipefd1[1]);
59
60          /* replace stdin with read end of pipefd2 */
61          if (dup2(pipefd1[0], 0) == -1) {
62      perror("dup2");
63      exit(EXIT_FAILURE);
64          }
65
66          /* close read end of pipefd2 */
67          close(pipefd2[0]);
68
69          /* replace stdout with write end of pipefd2 */
70          if (dup2(pipefd2[1], 1) == -1) {
71      perror("dup2");
72      exit(EXIT_FAILURE);
73          }
74
75          /* execute <command2> */
76          execlp( file: argv[2],  arg0: argv[2], (char *)NULL);
77          perror("execlp");
78          exit(EXIT_FAILURE);
```

```
79
80      } else if (pid2 < 0) { /* we have an error */
81        perror("fork"); /* use perror to print the system error message */
82        exit(EXIT_FAILURE);
83      }
84
85      pid3 = fork(); /* fork third process to execute command3 */
86      if (pid3 == 0) { /* this is child process */
87        /* close both ends of the pipefd1 */
88        close(pipefd1[0]);
89        close(pipefd1[1]);
90
91        /* close write end of pipefd2 */
92        close(pipefd2[1]);
93
94        /* replace stdin with read end of pipefd2 */
95        if (dup2(pipefd2[0], 0) == -1) {
96      perror("dup2");
97      exit(EXIT_FAILURE);
98        }
99
100       /* execute <command3> */
101       execlp( file: argv[3],  arg0: argv[3], (char *)NULL);
102       perror("execlp");
103       exit(EXIT_FAILURE);
104
105     } else if (pid3 < 0) { /* we have an error */
106       perror("fork"); /* use perror to print the system error message */
107       exit(EXIT_FAILURE);
108     }
109
110     /* close the pipes in the parent */
111     close(pipefd1[0]);
112     close(pipefd1[1]);
113     close(pipefd2[0]);
114     close(pipefd2[1]);
115
116     /* wait for both child processes to terminate */
117     waitpid(pid1, &status1, 0);
118     waitpid(pid2, &status2, 0);
119     waitpid(pid3, &status3, 0);
120
121     return 0;
122 }
```

compile & run

```
(base) mahmutunan@MacBook-Pro lecture29 % gcc -Wall pipe3.c -o pipe3

(base) mahmutunan@MacBook-Pro lecture29 % ./pipe3 ps sort wc
       5      23     161
(base) mahmutunan@MacBook-Pro lecture29 % _
```

pager.c

```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 */
  FILE *fp;
  char line[BUFSIZ];
  int n;

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

  if ( (fp = fopen( filename: argv[1], mode: "r")) == NULL) {
    printf("Error opening file %s for reading\n", argv[1]);
    exit(-1);
  }

  if (pipe(pipefd) == 0) { /* Open a pipe */
    if ((pid = fork()) == 0) { /* I am the child process */
      close(pipefd[1]);    /* close write end */
      dup2(pipefd[0], STDIN_FILENO); /* replace stdin of child */
      execlp( file: "/usr/bin/more", arg0: "more", (char *)NULL);
      perror("exec");
      exit(EXIT_FAILURE);
```

```
34          exit(EXIT_FAILURE);
35        } else if (pid > 0) { /* I am the parent process */
36          close(pipefd[0]);   /* close read end */
37          /* read lines from the file and write it to pipe */
38          while (fgets(line, BUFSIZ, fp) != NULL) {
39          n = strlen(line);
40              if (write( fd: pipefd[1], line, n) != n) {
41            printf("Error writing to pipe\n");
42            exit(-1);
43          }
44          }
45          close(pipefd[1]);   /* close write end */
46
47          wait(&status);      /* wait for child to terminate */
48          if (WIFEXITED(status))
49              printf("Child process exited with status = %d\n", WEXITSTATUS(status));
50          else
51              printf("Child process did not terminate normally!\n");
52        } else { /* we have an error in fork */
53          perror("fork");
54          exit(EXIT_FAILURE);
55        }
56      } else {
57        perror("pipe");
58        exit(EXIT_FAILURE);
59      }
60
61      exit(EXIT_SUCCESS);
62  }
```

compile & run

```
(base) mahmutunan@MacBook-Pro lecture29 % ./pager smalltale.txt
it was the best of times it was the worst of times
it was the age of wisdom it was the age of foolishness
it was the epoch of belief it was the epoch of incredulity
it was the season of light it was the season of darkness
it was the spring of hope it was the winter of despair
we had everything before us we had nothing before us
we were all going direct to heaven we were all going direct
the other wayin short the period was so far like the present
period that some of its noisiest authorities insisted on its
being received for good or for evil in the superlative degree
of comparison only

there were a king with a large jaw and a queen with a plain face
on the throne of england there were a king with a large jaw and
a queen with a fair face on the throne of france  in both
```

popen and pclose functions

- As we have seen in the examples, the common usage of pipes involve creating a pipe, creating a child process with fork, closing the unused ends of the pipe, execing a command in the child process, and waiting for the child process to terminate in the parent process.
- Since this is such a common usage, UNIX systems provide *popen* and *pclose* functions that perform most of these operations in a single operation.
- The C APIs for the *popen* and *pclose* functions are shown below:

```
#include <stdio.h>

FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

- The popen function performs the following steps:
- creates a pipe
- creates a new process using fork
- perform the following steps in the child process
  - close unused ends of the pipe (based on the *type* argument)
  - execs a shell to execute the *command* provided as argument to popen (i.e., executes "sh -c command")
- perform the following steps in the parent process
  - close unused ends of the pipe (based on the *type* argument)
  - wait for the child process to terminate

- The *popen* function returns the FILE handle to the pipe created so that the calling process can read or write to the pipe using standard I/O system calls.
- If the *type* argument is specified as read-only ("r") then the calling process can read from the pipe, this results in reading from the *stdout* of the child process (see Figure 15.9).
- If the type argument is specifies as write-only ("w") then the calling process can write to the pipe, this results in writing to the *stdin* of the child process created (see Figure 15.10).

- The FILE handle returned by *popen* must be closed using *pclose* to make sure that the I/O stream opened to read or write to the pipe is closed and wait for the child process to terminate.
- The termination status of the shell started by exec will be returned when the *pclose* function returns.
-

pipe2a.c

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    FILE *fp1, *fp2;
    char line[BUFSIZ];

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

    /* create a pipe, fork/exec command argv[1], in "read" mode */
    /* read mode - parent process reads stdout of child process */
    if ((fp1 = popen(argv[1], "r")) == NULL) {
    perror("popen");
    exit(EXIT_FAILURE);
    }

```

```
19
20      /* create a pipe, fork/exec command argv[2], in "write" mode */
21      /* write mode - parent process writes to stdin of child process */
22      if ((fp2 = popen(argv[2], "w")) == NULL) {
23      perror("popen");
24      exit(EXIT_FAILURE);
25      }
26
27      /* read stdout from child process 1 and write to stdin of
28         child process 2 */
29      while (fgets(line, BUFSIZ, fp1) != NULL) {
30      if (fputs(line, fp2) == EOF) {
31         printf("Error writing to pipe\n");
32         exit(EXIT_FAILURE);
33          }
34      }
35
36      /* wait for child process to terminate */
37      if ((pclose(fp1) == -1) || pclose(fp2) == -1) {
38      perror("pclose");
39      exit(EXIT_FAILURE);
40      }
41
42      return 0;
43  }
44
```

compile & run

```
(base) mahmutunan@MacBook-Pro lecture29 % ./pipe2a "ls -l" sort
-rw-r--r--@ 1 mahmutunan  staff    105 Nov  2 13:37 p1.c
-rw-r--r--@ 1 mahmutunan  staff    169 Nov  2 13:36 p2.c
-rw-r--r--@ 1 mahmutunan  staff    790 Oct 27 22:41 popen.c
-rw-r--r--@ 1 mahmutunan  staff   1694 Oct 27 22:41 pager2.c
-rw-r--r--@ 1 mahmutunan  staff   1853 Nov  4 13:07 pipe2a.c
-rw-r--r--@ 1 mahmutunan  staff   2073 Oct 27 22:41 pipe1.c
-rw-r--r--@ 1 mahmutunan  staff   2121 Oct 27 22:41 pipe0.c
-rw-r--r--@ 1 mahmutunan  staff   2284 Nov  4 12:44 pager.c
-rw-r--r--@ 1 mahmutunan  staff   2782 Nov  4 11:31 pipe2.c
-rw-r--r--@ 1 mahmutunan  staff   3858 Nov  4 11:51 pipe3.c
-rw-r--r--@ 1 mahmutunan  staff   5074 Nov  4 12:51 smalltale.txt
-rwxr-xr-x  1 mahmutunan  staff  12556 Nov  2 13:37 p1
-rwxr-xr-x  1 mahmutunan  staff  12604 Nov  2 13:37 p2
-rwxr-xr-x  1 mahmutunan  staff  12952 Nov  4 13:07 pipe2a
-rwxr-xr-x  1 mahmutunan  staff  12984 Nov  2 13:37 pipe0
-rwxr-xr-x  1 mahmutunan  staff  12996 Nov  2 13:20 pipe1
-rwxr-xr-x  1 mahmutunan  staff  13040 Nov  4 11:31 pipe2
-rwxr-xr-x  1 mahmutunan  staff  13040 Nov  4 12:41 pipe3
-rwxr-xr-x  1 mahmutunan  staff  13076 Nov  4 12:44 pager
total 352
```

- Note that since the command is executed using a shell, we can provide wildcards and other special characters that the shell can expand.
- Also note that in this version of the program the parent process is reading the *stdout* stream of the first child process and then writing to the *stdin* stream of the second child process (we did not do this in the first version).

Here is an updated version of the pager program that uses popen and pclose

pager2.c

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
   FILE *fpin, *fpout;
   char line[BUFSIZ];

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

   /* open file for reading */
   if ( (fpin = fopen( filename: argv[1],  mode: "r")) == NULL ) {
     printf("Error opening file %s for reading\n", argv[1]);
     exit(-1);
   }

   /* create a pipe, fork/exec process "more", in "write" mode */
   /* write mode - parent process writes, child process reads */
   if ( (fpout = popen("more", "w")) == NULL ) {
       perror("exec");
       exit(EXIT_FAILURE);
   }
```

```c
    /* read lines from the file and write it fpout */
    while (fgets(line, BUFSIZ, fpin) != NULL) {
        if (fputs(line, fpout) == EOF) {
            printf("Error writing to pipe\n");
      exit(EXIT_FAILURE);
        }
    }

    /* close the pipe and wait for child process to terminate */
    if (pclose(fpout) == -1) {
        perror("pclose");
        exit(EXIT_FAILURE);
    }

    exit(EXIT_SUCCESS);
}
```

compile & run

```
(base) mahmutunan@MacBook-Pro lecture29 % gcc -Wall pager2.c -o page
r2
(base) mahmutunan@MacBook-Pro lecture29 % ./pager2 smalltale.txt
it was the best of times it was the worst of times
it was the age of wisdom it was the age of foolishness
it was the epoch of belief it was the epoch of incredulity
it was the season of light it was the season of darkness
it was the spring of hope it was the winter of despair
we had everything before us we had nothing before us
we were all going direct to heaven we were all going direct
the other wayin short the period was so far like the present
```

popen.c

- You can also find a simpler version of the program that uses a single popen system call to create a pipe in "read" mode, execute the command specified as the command-line argument, reads the pipe and prints it to stdout

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    FILE *fp;
    char line[BUFSIZ];

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

```
    if ((fp = popen(argv[1], "r")) == NULL) {
    perror("popen");
    exit(EXIT_FAILURE);
    }

    while (fgets(line, BUFSIZ, fp) != NULL) {
      fputs(line, stdout);
    }

    if (pclose(fp) == -1) {
    perror("pclose");
    exit(EXIT_FAILURE);
    }

    return 0;
}
```

compile & run

```
(base) mahmutunan@MacBook-Pro lecture29 % gcc -Wall popen.c -o popen

(base) mahmutunan@MacBook-Pro lecture29 % ./popen ps
  PID TTY           TIME CMD
 1600 ttys000    0:00.46 -zsh
26658 ttys000    0:00.00 ./popen ps
(base) mahmutunan@MacBook-Pro lecture29 %
```