Operating Systems Lab - Week 8: exercise
- with answers

In this lab, you will implement three programs that create a message-passing half-duplex pipe between processes that *have a common ancestor*:

1. `child2parent.c`, where the child sends formatted messages to its parent, as in `createPipe.c` ,
2. `parent2child.c`, where the parent sends formatted messages to its child, and
3. `child2child.c`, where the parent generates two children and the first child sends messages to the second child.

`createPipe.c` implements a *IPC channel* between a child and its parent by calling `pipe()` [1], which creates an *anonymous pipe* in the calling program. After calling `pipe()`, the program generates a child process with `fork()`. The *file descriptors* of the pipe ends are copied into the child address space and the two processes can communicate by *writing to* and *reading from* it. As *half-duplex pipes* are one-way message-passing channels, you must *choose* the direction of the information flow and close the *unused end of the pipe* in each process. For example, in `createPipe.c` , the parent can *only* read from and the child can only write into the pipe.

Three similar programs. Obtain `child2parent.c` by modifying `createPipe.c` . You only need a new implementation of the *reading* and *writing* subroutines. `parent2child.c` works in the opposite way. Assign the channel ends differently to make the parent write and the child read. Again the program structure will be very similar to `createPipe.c` . To write `child2child.c`, modify the main function because the parent will generate *two* children after creating the pipe. The first child will write messages into the pipe and the second will read the messages sent by the first. The parent will not interfere with the children's conversation and will close immediately both ends of the pipe.

Formatted messages. Instead of sending and receiving fixed hard-coded messages, the programs will convert the user `stdin` input into formatted messages, i.e. integers. In particular, the *writing process*

- reads a string from the terminal,
- separates the string into words, i.e. groups of characters ending with `' '`,
- processes the words one by one to check whether they contain *numerical characters*, i.e. `'0', '1', ..., '9'`,
- removes all *non-numerical* characters and converts the word into the corresponding integer, e.g. `one12two` will be converted into the integer 12, and
- send the obtained integer to the *reading process*, through the pipe.

The *reading process*, i.e. the child in `parent2child.c` and the second child in `child2child`

- reads the integers sent by the other process,
- computes their sum, and
- print the obtained value on the terminal.

Sending formatted messages through the pipe using the *high-level* I/O functions defined in `stdio.h` require referring to the pipe ends through pointers to their *file handle*, i.e. pointers to `struct`-objects of type `FILE`. You can obtain the pointers to the file handles of the pipe by calling

```
FILE *pReading = fdopen(fd[0], "r");
FILE *pWriting = fdopen(fd[1], "w");
```

[1] Include `unistd.h` in your program header.

1 child2parent.c

In this section, you will write a program, `child2parent.c`, where:

- an anonymous pipe is created by calling `pipe()`,
- a child process and a parent process are created by calling `fork()`,
- the child process converts an input string of words into a series of integers, e.g.

  ```
  ... one 1 two2 three34four and 5five6six7seven   ...
  ```

  will produce 0, 0, 1, 2, 34, 0, 567, and 0,
- the child sends the obtained integers (iteratively) to the parent as separate *formatted* messages through the pipe by calling

  ```
  fprintf(pWriting, "%d ",n)
  ```

  where `n` is the integer associated with the current word,
- the child sends a negative integer, e.g. `n` $= -1$ to tell the parents that the previous one was the last valid number obtained from the input,
- the parent reads the integers sent by the child by calling

  ```
  fscanf(pReading, "%d",n)
  ```

  ‘
- the parent computes the sum of the received messages by updating an integer variable `sum` through

  ```
  if (n >= 0) sum = sum + n;
  ```
- the parent prints the sum on the terminal before exiting.

1.1 Create a *high-level* pipe

Normally, when you create a pipe by calling `pipe()`, you refer to its ends through the corresponding *file descriptors*. In this section, you see how you can handle the pipe by using the corresponding *file handles*. As in `createPipe.c` , start by creating a 2-entry integer array declared as

```
int fd[2];
```

whose entries will be loaded with the file descriptors associated with the pipe's *reading* and *writing* ends(in this order). To create the new channel, write

```
pipe(fd);
```

exactly as in `createPipe.c` . Check whether the pipe has been created successfully by looking at the value of the return value of `pipe` and the entries of `fd`. To obtain the file handles associated with the file descriptors loaded in `fd`, write

```
FILE *pReading = fdopen(fd[0], "r");
FILE *pWriting = fdopen(fd[1], "w");
```

with `fdopen` being defined in `stdio.h`. See Section 13.4 of the GNU online manual for more details about `fdopen` and other similar functions. Note that you will need `pReading` and `pWriting` to use the *formatted I/O functions* defined in `stdio.h`, e.g. `fprintf` and `fscanf`.

1.2 Create a child process and close the unused ends of the pipe

The next step is to call `fork` and create a child process that will inherit both pointers to the pipe file handles, `pReading` and `pWriting`. In the child, who will be sending the messages, close the *reading* end of the pipe by calling

```
fclose(pReading);
```

In the parent, who will read the child's messages, close the *writing* end of the pipe by calling

```
fclose(pWriting);
```

Note that you should use `fclose` instead of `close` because `pReading` and `pWriting` are pointers to file handles. See Section 12.4 of the GNU online manual for more details about `fclose` and other similar functions.

1.3 Define the child and the parent processes

The child can now send messages to the parent through the pipe by writing on the *"file"* pointed by `pWriting`. To select the child process in your code, introduce an `if (!fork())-else` conditional block.

The child process. In the `if`-part of the conditional block, call the writing subroutine

```
int writeMessage(FILE *pf, char *author);
```

with the correct value of the first argument and the string `"the child"` as the second argument. As a writing subroutine, you can use

```
int writeMessage(FILE *pf, char *author) {
  printf("enter integers\n");
  char c = '\0';
  while (c != '\n') {
    int n = 0;
    while((c = getchar()) != ' ' && c != '\n')
      if (c<= '9' && c >= '0')
        n = n * 10 + c - '0';
    printf("%s writes to fd[1]: %d \n", author, n);
    fprintf(pf, "%d ", n);
  }
  fprintf(pf, "%d", -1);
  return 0;
}
```

When the writing subroutine returns the child has finished and can

- close the writing end of the pipe by calling

  ```
  fclose(pWriting);
  ```

- exit.

The parent process. In the `else`-part or the conditional block, call the writing subroutine

```
int readMessage(FILE *pf, char *author);
```

with the correct value of the first argument and the string `"the parent"` as the second argument. As a reading subroutine, you can use

```
int readMessage(FILE *pf, char *reader) {
  int i = 0, sum = 0;
  while(i != -1) {
    fscanf(pf, "%d", &i);
    if (i != -1) {
```

```
      sum = sum + i;
      printf("%s read from fd[0]: %d \n", reader, i);
    }
  }
  return sum;
}
```

After the reading subroutine returns, the parent has *almost* finished its job and should

- print on `stdout` the sum of the received messages, i.e. the return value of `readMessage` by calling

  ```
  printf("sum=%d\n", sum);
  ```

- close the reading end of the pipe by calling

  ```
  fclose(pReading);
  ```

- and exit.

1.4 Expected output

On `linux.cim.rhul.ac.uk`, your program should produce an output analogous to

```
enter integers
one 1 two 2 three3four 45five    s i x 6 85eigthyfive
the child writes to fd[1]: 0
the child writes to fd[1]: 1
the child writes to fd[1]: 0
the child writes to fd[1]: 2
the child writes to fd[1]: 3
the child writes to fd[1]: 45
the child writes to fd[1]: 0
the child writes to fd[1]: 0
the child writes to fd[1]: 0
the child writes to fd[1]: 0
the child writes to fd[1]: 0
the child writes to fd[1]: 0
the child writes to fd[1]: 6
the child writes to fd[1]: 85
the parent read from fd[0]: 0
the parent read from fd[0]: 1
the parent read from fd[0]: 0
the parent read from fd[0]: 2
the parent read from fd[0]: 3
the parent read from fd[0]: 45
the parent read from fd[0]: 0
the parent read from fd[0]: 0
the parent read from fd[0]: 0
the parent read from fd[0]: 0
the parent read from fd[0]: 0
the parent read from fd[0]: 0
the parent read from fd[0]: 6
the parent read from fd[0]: 85
sum=142
```

Answer:

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

```
int readMessage(FILE *pf, char *reader);
int writeMessage(FILE *pf, char *author);
int main() {
  int fd[2];
  pipe(fd);
  FILE *pReading = fdopen(fd[0], "r");
  FILE *pWriting = fdopen(fd[1], "w");
  if (!fork()) {
    fclose(pReading);
    writeMessage(pWriting, "the child");
    fclose(pWriting);
    return 0;
  }
  else {
    fclose(pWriting);
    int sum  = readMessage(pReading, "the parent");
    printf("sum=%d\n", sum);
    fclose(pReading);
  }
}

int writeMessage(FILE *pf, char *author) {
  printf("enter integers\n");
  char c = '\0';
  while (c != '\n') {
    int n = 0;
    while((c = getchar()) != ' ' && c != '\n') {
      if (c<= '9' && c >= '0')
        n = n * 10 + c — '0';
    }
    printf("%s writes to fd[1]: %d \n", author, n);
    fprintf(pf, "%d ", n);
  }
  fprintf(pf, "%d", —1);
  return 0;
}
int readMessage(FILE *pf, char *reader) {
  int i = 0, sum = 0;
  while(i != —1) {
    fscanf(pf, "%d", &i);
    if (i != —1) {
      sum = sum + i;
      printf("%s read from fd[0]: %d \n", reader, i);
    }
  }
  return sum;
}
```

2 parent2child.c

Change a few details of `child2parent.c` so that

- the child reads and computes the sum of the messages sent by the parent and
- the parent converts the user input into integers and sends the obtained integers to the child

2.1 Expected output

On `linux.cim.rhul.ac.uk`, your program should produce an output analogous to

```
enter integers
  one1 two23three 4fourfiveSix56
the child writes to fd[1]: 0
the child writes to fd[1]: 0
the child writes to fd[1]: 1
the child writes to fd[1]: 23
the child writes to fd[1]: 456
the child writes to fd[1]: 0
the child writes to fd[1]: 0
the child writes to fd[1]: 0
the child writes to fd[1]: 0
the parent read from fd[0]: 0
the parent read from fd[0]: 0
the parent read from fd[0]: 1
the parent read from fd[0]: 23
the parent read from fd[0]: 456
the parent read from fd[0]: 0
the parent read from fd[0]: 0
the parent read from fd[0]: 0
the parent read from fd[0]: 0
sum=480
```

Answer:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int readMessage(FILE *pf, char *reader);
int writeMessage(FILE *pf, char *author);
int main() {
  int fd[2];
  pipe(fd);
  FILE *pReading = fdopen(fd[0], "r");
  FILE *pWriting = fdopen(fd[1], "w");
  if (!fork()) {
    fclose(pReading);
    writeMessage(pWriting, "the child");
    fclose(pWriting);
    return 0;
  }
  else {
    fclose(pWriting);
    int sum  = readMessage(pReading, "the parent");
    printf("sum=%d\n", sum);
    fclose(pReading);
  }
}

int writeMessage(FILE *pf, char *author) {
  printf("enter integers\n");
  char c = '\0';
  while (c != '\n') {
    int n = 0;
    while((c = getchar()) != ' ' && c != '\n') {
      if (c<= '9' && c >= '0')
        n = n * 10 + c - '0';
```

```
    }
    printf("%s writes to fd[1]: %d \n", author, n);
    fprintf(pf, "%d ", n);
  }
  fprintf(pf, "%d", -1);
  return 0;
}
```

3 child2child.c

Change `child2parent.c` again so that

- the parent generates *two children*,
- the first child converts the user input into integers and sends the obtained integers to the child, and
- the second child reads and computes the sum of the messages sent by the first child.

3.1 Structure of `main`

You can use the following template for `main`.

```
int main() {
  ...
  for (int i = 0; i<2; i++) {
    if (i == 0) {
      if (!fork()) {
        ...
        return 0;
      }
    }else{
      if (!fork()) {
        ...
        return 0;
      }
    }
    wait(NULL);
  }
}
```

3.2 Expected output

On `linux.cim.rhul.ac.uk`, your program should produce an output analogous to

```
enter integers
1    an d 2two three34 four    5fiveAndsix6
the writing child writes to fd[1]: 1
the writing child writes to fd[1]: 0
the writing child writes to fd[1]: 0
the writing child writes to fd[1]: 0
the writing child writes to fd[1]: 0
the writing child writes to fd[1]: 0
the writing child writes to fd[1]: 2
the writing child writes to fd[1]: 34
the writing child writes to fd[1]: 0
the writing child writes to fd[1]: 0
the writing child writes to fd[1]: 0
the writing child writes to fd[1]: 0
```

```
the writing child writes to fd[1]: 56
the reading child read from fd[0]: 1
the reading child read from fd[0]: 0
the reading child read from fd[0]: 0
the reading child read from fd[0]: 0
the reading child read from fd[0]: 0
the reading child read from fd[0]: 0
the reading child read from fd[0]: 2
the reading child read from fd[0]: 34
the reading child read from fd[0]: 0
the reading child read from fd[0]: 0
the reading child read from fd[0]: 0
the reading child read from fd[0]: 0
the reading child read from fd[0]: 56
sum=93
```

Answer:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#define MAXCHARS 100
int readMessage(FILE *pf, char *reader);
int writeMessage(FILE *pf, char *author);

int main() {
  int fd[2];
  pipe(fd);
  FILE *pReading = fdopen(fd[0], "r");
  FILE *pWriting = fdopen(fd[1], "w");
  for (int i = 0; i<2; i++) {
    if (i == 0) {
      if (!fork()) {
        fclose(pReading);
        writeMessage(pWriting, "the writing child");
        fclose(pWriting);
        return 0;
    }
    }else{
      if (!fork()) {
        fclose(pWriting);
        int sum  = readMessage(pReading, "the reading child");
        printf("sum=%d\n", sum);
        fclose(pReading);
        return 0;
      }
    }
    wait(NULL);
  }
}
int writeMessage(FILE *pf, char *author) {
  printf("enter integers\n");
  char c = '\0';
  while (c != '\n') {
    int n = 0;
    while((c = getchar()) != ' ' && c != '\n') {
      if (c >= '0' && c <= '9')
        n = n * 10 + c — '0';
    }
```

```
    printf("%s writes to fd[1]: %d \n", author, n);
    fprintf(pf, "%d ", n);
  }
  fprintf(pf, "%d ", —1);
  return 0;
}
int readMessage(FILE *pf, char *reader) {
  int i = 0, sum = 0;
  while(i != —1) {
    fscanf(pf, "%d", &i);
    if (i != —1) {
      sum = sum + i;
      printf("%s read from fd[0]: %d \n", reader, i);
    }
  }
  return sum;
}
```