Seminary 4

IPC (Inter Process Communication)

IPC methods in Unix systems: `<unistd.h>`

Pipe **FIFO (or named pipe)** **Shared memory**	**Semaphores** **Socket** **Message queues**

Pipe/FIFO

Unidirectional flow of bytes between two processes: one writes bytes in the pipe/fifo, the other reads from it, in a first in first out order.

PIPE

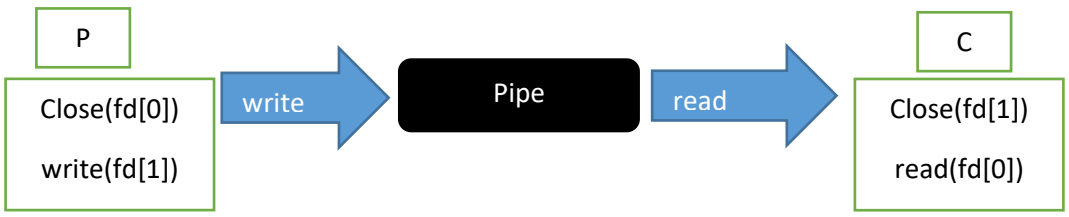
```
int pipe(int fd[2]);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

`fd[0]` – reading end of the fd pipe

`fd[1]` – writing end of the fd pipe

Programmer must ensure that the flow of bytes is unidirectional. For this we close the pipe ned that are not used in a process: close(fd[0]) for the process that only writes and close(fd[1]) for the process that only reads from the pipe.

Pipe is used between related processes, for example Parent write to a Child process. A pipe does not exist outside the 'life' of a process, so only a process that was created by the same process (with fork) that created the pipe can know about it and access it.



1. Adding four numbers in parallel. Child process adds first 2, parent last two and centralises the result. Close pipe ends that are not used as soon as you can close them!

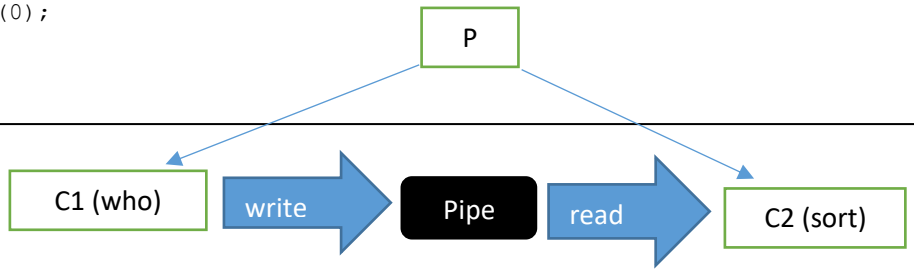
```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
int main () {
    int a[] = {1,2,3,4}, f[2];
    pipe(f);
    if (fork()==0) { // Child process
        close(f[0]); // child doe not read from the pipe, only write to it
        a[0]+=a[1];
        write(f[1], &a[0], sizeof(int)); //Write sum of first two numbers
        close(f[1]); //close the pipe and terminate the process
        exit(0);
    }
    close(f[1]); // Parent process does not write, just reads
    a[2]+=a[3]; // Computes the sum of the last two numbers
    read(f[0], &a[0], sizeof(int)); // read  the patial sum of first two no
    close(f[0]); // done reading, close it
    wait(NULL); //wait for the child to finish
    a[0]+=a[2]; // use the patial sums to compute the total sum
    printf("Suma este %d\n", a[0]);
    return(0);
}
```

DUP2

`int dup2 (int fd_f, int fd_r); //Redirect from fd_r to fd_f;`

2. Write a program to execute who |sort by creating two child processes that each launch who and sort and redirect their input/output accordingly.

```
...
main (){
       int p[2];
       pipe (p);
       if (fork () == 0) { // First child
              dup2 (p[1], 1); //Redirect standard output (1 or stdout) to a pipe
              close (p[0]);
              execlp ("who", "who", 0); //launch program who
              //exit(1); in case of error with exec
       }
       else if (fork () == 0) { // Second child
              dup2 (p[0], 0); // Redirect standard input (0 or stdin) from pipe
              close (p[1]);
              execlp ("sort", "sort", 0); //execute sort, input read from pipe
              //exit(1); if error with exec...
       }
       else { // Parent process
              close (p[0]);
              close (p[1]);
              wait (0);
              wait (0);
       }
       return 0;
}
```



POPEN

```
#include <stdio.h>
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

command – is a string containing a shell cmd; type- “r” or “w” if we read or write from/to pipe

Popen **opens a pipe and executes a fork which opens a shell**. The child process executes the command and:

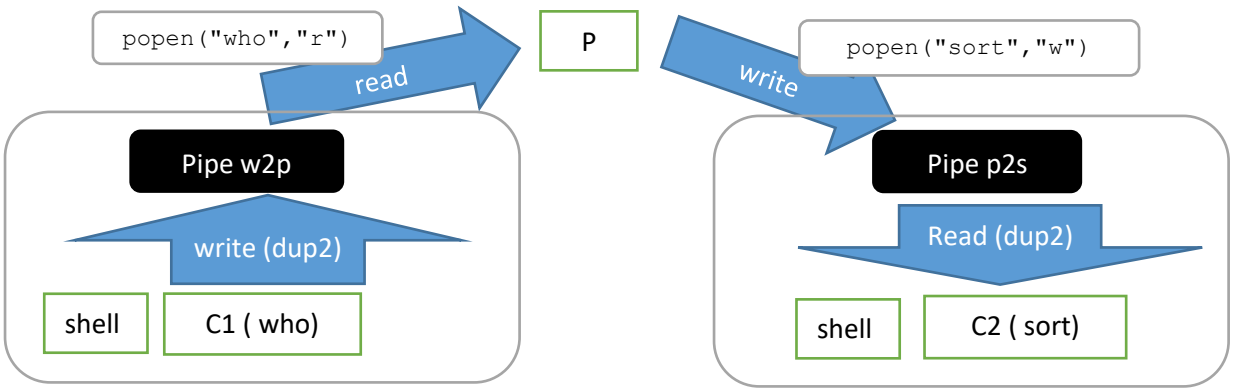
- For “r” the calling process gets a file pointer to read from the pipe the standard output of the command, the fd[0] end of the pipe for reading, while the other end is redirecting the output of the command from `stdout` or 1 to the pipe (equivalent of `dup2(fd[1], 1))`
- For “w” the calling process gets a file pointer to write in the pipe the standard input for the command, the fd[1] end of the pipe for writing, while the other end is duplicating the `stdin` or 0 (equivalent of `dup2(fd[0], 0)`)

```
Popen:
Pipe + fork + shell exec + dup
```

3. Write a program to execute who |sort using `popen`. We will need two pipes with popen because they can not be used two ways (either r or w). First pipe w2p will contain the output of who command executed and then we will read the content of the pipe w2p and write it into the second pipe p2s as an input for command sort. The result from sort would be on the stdout.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(){
        FILE *w2p; //who to parent
        FILE *p2s; //parent to sort
        W2p=popen("who","r");
        p2s=popen("sort","w");
        char  line[50];

        while (fgets(line,50,w2p)){
                fprintf(p2s,"%s",line);
        }
        pclose(w2p);
        pclose(p2s);
        return 0;
}
```



Each popen creates a pipe and a child process that uses exec to run the command. We have two child processes, and they communicate only with the parent process (not with each other).

The problem can be solved even simpler.

```
FILE* p=popen("who  |  sort","r");

// and then read from the pipe p the output result of the who|sort and print
on stdout

while (fgets(line,50,p)){  printf("%s",line);}
```

When executing the three programs above with `ps | sort` (instead of who | sort) we get the output:

C program	Output	Internal mechanism
Dup2+fork+ exec	6 tty1 00:00:01 bash 365 tty1 00:00:00 a.out 366 tty1 00:00:00 ps 367 tty1 00:00:00 sort PID TTY TIME CMD	Processes (3): - Initial C program - Fork+exec(replaced by)-> ps - Fork+exec(replaced by)-> sort Pipes (1): p
Popen (ps), popen (sort)	6 tty1 00:00:00 bash 353 tty1 00:00:00 a.out 354 tty1 00:00:00 sh 355 tty1 00:00:00 sh 356 tty1 00:00:00 ps 357 tty1 00:00:00 sort PID TTY TIME CMD	Processes (5): - Initial C program - Popen performes fork and launches shell that executes ps (2 processes: sh+ ps) - Popen performes fork and launches shell that executes sort (2 processes: sh + sort) Pipes (2): - ps to main program - main program to sort
Popen (ps | sort)	6 tty1 00:00:01 bash 401 tty1 00:00:00 a.out 402 tty1 00:00:00 sh 403 tty1 00:00:00 ps 404 tty1 00:00:00 sort PID TTY TIME CMD	Processes (4): - Initial C program - Popen performes fork an launches shell that executes both ps and sort using a shell pipe (3 processes) Pipes (1): - From Ps| sort to main program

Compare this with the output of executing: $ `ps & sh capit.sh exemplu.txt`

Where capit.sh is a shell script containing:

```
#!/bin/sh

sed "s/\<\([a-z]\)/\u\1/g" $1 > ${1}_temp
```

Output clearly contains the ps process running the background and two processes: the shell and sed command from the script.

```
[1] 490
  PID TTY          TIME CMD
    6 tty1     00:00:01 bash
  490 tty1     00:00:00 ps
  491 tty1     00:00:00 sh
  492 tty1     00:00:00 sed
[1]+  Done                    ps
```

FIFO - first-in first-out special file, named pipe

```
#include <fcntl.h>
#include <sys/stat.h>
int mknod(const char *pathname, mode_t mode, dev_t dev);
int mkfifo(const char *pathname, mode_t mode);
int unlink(const char *pathname);
```

or command line

```
$ mknod pathname p
$ mkfifo pathname
$ rm pathname
```

Where pathname is the name of the fifo. FIFO is a stream that exists outside a specific process, so it can be used for communication between processes that are not related, unlike pipe. Usually a fifo in created in the `/tmp` directory `$ mkfifo /tmp/fifo1`. Delete with `$ rm /tmp/fifo1`.

4. Add four numbers in parallel using FIFO. Use two related processes.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <sys/types.h>
int main () {
    int a[] = {1,2,3,4}, f;
    //mkfifo("/tmp/fifo1", 0666);  // if not created in the cmd line
    if (fork()==0) { // Child process
        f = open("/tmp/fifo1", O_WRONLY);
        a[0]+=a[1];
        write(f, &a[0], sizeof(int)); // Write partial sum
        close(f);
        exit(0);
    }
    f = open("/tmp/fifo1", O_RDONLY); //parent process
    a[2]+=a[3];
    read(f, &a[0], sizeof(int));
    close(f);
    wait(NULL);
    a[0]+=a[2];
    printf("Suma este %d\n", a[0]);
    //unlink("/tmp/fifo1"); //if we create fifo in the code, destroy also
    return 0;
}
```

5. Add four numbers in parallel using FIFO. Processes are created by different sources. This time we create our fifo in the command line and then we run each program. The order is not important, the open/read calls are blocking and will wait for each other.

ProgramSimilartoParent.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
int main () {
    int a[] = {1,2,3,4}, f;
    f=open("/tmp/fifo1",O_RDONLY);
    a[2]+=a[3];
    read(f,&a[0],sizeof(int));
    close(f);
    a[0]+=a[2];
    printf("Suma este %d\n", a[0]);
}
```

ProcessSimilartoChild.c

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <sys/types.h>
int main () {
    int a[] = {1,2,3,4}, f;
    f=open("/tmp/fifo1", O_WRONLY);
    a[0]+=a[1];
    write(f, &a[0], sizeof(int));
    close(f);
}
```

Other examples with FIFO:

a. Send byte buffers, prefixed by their length. If sending strings in this way, it is a good idea to send the ending zero.

```
// WRITER
int main(int argc, char**argv) {
   int f, n;
   char* s = "Hello!";

   f = open("w2r", O_WRONLY);
   n = strlen(s)+1;
   write(f, &n, sizeof(int));
   write(f, s, n);
   close(f);

   return 0;
}
```

```
// READER
int main(int argc, char**argv) {
   int f, n;
   char* s;

   f = open("w2r", O_RDONLY);
   read(f, &n, sizeof(int));
   s = (char*)malloc(n);
   read(f, s, n);
   free(s);
   close(f);

   return 0;
}
```

b. Send structures as byte buffers.
   i. The length prefix is not required, as we know the length of the message.
   ii. If field `b` inside `abc` would be declared as `char*`, and allocated dynamically, this solution would send only the address `b`, not its content, which in the reader does not mean anything.

```
// header.h
typedef struct {
   int a;
   char b[10];
   float c;
} abc;
```

```
// WRITER
#include "header.h"
int main() {
   int f;
   abc x;

   x.a = 5;
   strcpy(x.b, "qwerty");
   x.c = 1.0f;

   f = open("w2r", O_WRONLY);
   write(f, &x, sizeof(abc));
   close(f);

   return 0;
}
```

```
// READER
#include "header.h"
int main() {
   int f;
   abc x;

   f = open("w2r", O_RDONLY);
   read(f, &x, sizeof(abc));
   close(f);

   return 0;
}
```

Retrying read/write operations on pipe/FIFO

a. Both read and write will return the number of bytes they processed (i.e. read or written). If that number is less than what we asked, it may mean the data has not arrived yet, or there is no more space in the pipe/FIFO. This is very common when the size of the data is larger.

b. A simple way to do this is to write a function that tries to read/write a given number of times.

```
int stubborn_read(int fd, void* buf, int count, int trials) {
   int k, total = 0, n = 0;

   while(total < count && n < trials && (k=read(fd, buf+total, count-total)) > 0) {
      total += k;
      n++;
   }
   return k < 0 ? k : total;
}
```

```
int stubborn_write(int fd, void* buf, int count, int trials) {
   int k, total = 0, n = 0;

   while(total < count && n < trials && (k=write(fd, buf+total, count-total)) > 0) {
      total += k;
      n++;
   }
   return k < 0 ? k : total;
}
```

IMPORTANT

Open is a blocking call! The FIFO must be opened on both ends (reading and writing) before data can be passed. Normally, opening the FIFO blocks until the other end is opened also.

When we call open for reading a fifo in a process, the execution will stop at that points until the fifo in opened by another process for writing. When using more fifos, make sure to open them in the same order in both processes to avoid deadlocks.

Read / write are also blocking calls!

Please refer to the linux manual for more information on using open with FIFOs. (see fifo, open, read, write in sections 1, 2, 3, 7)

DEADLOCK

A minimal example which deadlocks (first process writes to a and b, second process reads from a and b):

```
mkfifo a b
(> a; > b; ) &
(< b; < a; ) &
wait
```

Solution: open in the same order for both read and write:

```
(> a; > b; ) &
(< a; < b; ) &
wait
```

6. Change problem 4, such that the parent process, after receiving the partial sum from the child process, sends the final sum to the child using an additional fifo, and the child process reads it and prints it. Discuss about potential deadlocks.

Fifo a – from child to parent; fifo b - from parent to child;
Parent: <a; >b;
Child: > a; < b;

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

int main () {
        int nr[] = {1,2,3,4}, a, b;
        mkfifo("/tmp/fifoa", 0666); // we create fifo
        mkfifo("/tmp/fifob", 0666);

        if (fork()==0) { // Child process  ( > a; < b;) write to a, read from b
                a = open("/tmp/fifoa", O_WRONLY);
                b = open("/tmp/fifob", O_RDONLY);

                nr[0]+=nr[1];
                write(a, &nr[0], sizeof(int)); // Write partial sum
                read(b, &nr[0],sizeof(int)); //read total sum from parent

                close(a);
                close(b);

                printf("Child: Sum is: %d\n",nr[0]);
                exit(0);
        }
        a = open("/tmp/fifoa", O_RDONLY);//parent(<a; >b;)read from a, write to b
        b= open("/tmp/fifob", O_WRONLY); // use same oreder to open the fifos!!!
        //simulate deadlock: change order of opening a and b and we have deadlock
        //a=open("/tmp/fifoa", O_RDONLY);

        nr[2]+=nr[3];
        read(a, &nr[0], sizeof(int));
        close(a);

        nr[0]+=nr[2];
        write(b, &nr[0], sizeof(int)); //write total sum to child

        close(b);
        wait(0);
        printf("Parent finihed\n");
        unlink("/tmp/fifoa"); //when we have deadlock we must close with ctr+C
        unlink("/tmp/fifob"); //and destroy fifos in cmd line as code does not
//reach execution here and when running next we might get errors; same with open…

        return 0;
}
```

7. Given the following code, and that it executes successfully:

a) What will be printed?

b) How many processes are being created, including the initial process, if we comment line 8? Specify the hierarchy of the processes.

c) How many processes will be created if we move line 8 to line 11?

d) What will be printed if lines 16 and 17 will be moved inside the else clause starting line 11 of the initial code?

```
1   int main() {
2       int pfd[2], i, n;
3       pipe(pfd);
4       for(i=0; i<3; i++) {
5           if(fork() == 0) {
6               write(pfd[1], &i, sizeof(int));
7               close(pfd[0]); close(pfd[1]);
8               exit(0);
9           }
10          else {
11              // see c) and d)
12          }
13      }
14      for(i=0; i<3; i++) {
15          wait(0);
16          read(pfd[0], &n, sizeof(int));
17          printf("%d\n", n);
18      }
19      close(pfd[0]); close(pfd[1]);
20      return 0;
21  }
```

Solutions:

a) 0,1,2, on separate lines in any order

b) 8 processes, as a tree

c) 4 processes as a tree

d) 0,1,2, on separate line always in this order

8. Implement a simple `popen`/`pclose` API using `fork`, `exec`, and `dup2`.

a. Variable `caller_idx` is the pipe end to be returned to the caller. As the child process needs to do the opposite operation on the pipe (i.e. if the caller wants to read, the child process must write, and vice versa), the child process will close it. Variable `child_idx` is the other pipe end, which will be closed by the caller but used by the child process.

```
FILE* mypopen(char* cmd, char* type) {
        int p[2], caller_idx, child_idx;
        pipe(p);

        caller_idx = 0;
        if(type[0] == 'w') {
               caller_idx = 1;
        }
        child_idx = (caller_idx+1) % 2;

        if(fork() == 0) {
               close(p[caller_idx]);
               dup2(p[child_idx], child_idx);
               if(execlp("bash", "bash", "-c", cmd, NULL) < 0) {
                   close(p[child_idx]);
                   exit(1);
               }
        }

        close(p[child_idx]);
        return fdopen(p[caller_idx], type);
}
```

```
void mypclose(FILE* fd) {
     fclose(fd);
     wait(0);
}
```