Unix Pipe Communication

1. Suppose we want to speed-up calculation by parallelizing it using processes.
   The program below is a very simple implementation of the idea above but the result will be 6 i nstead of 10. Why? The child process stores a[2] in its own memory, and the parent process use the a[2] from its own memory, which does not have the child's calculation.

```
int main(int argc, char**argv) {
    int a[4] = {1, 2, 3, 4};
    int pid;

    pid = fork();
    if(pid == 0) {
        a[2] += a[3];
        exit(0);
    }
    a[0] += a[1];
    wait(0);
    a[0] += a[2];
    printf("%d\n", a[0]);
    return 0;
}
```

To fix this we need a way of communicating between processes. The easiest way to do this is using pipes.

a. A pipe is a buffer in memory that is opened like a file, twice, once for reading and once for writing. The opening is done automatically when we create it and we get back an array with the two int descriptors that result. The descriptor on position 0 is for reading, and the one on position 1 is for writing.

b. Special **read** behaviours
   i. Removes data from the pipe
   ii. If the pipe is empty, it waits for some data to arrive or for no writer to be connected to the pipe
   iii. It may not return all the data requested to be read, but it returns the lenght of what it actually read

c. Special **write** behaviours
   i. Adds data to the pipe
   ii. If the pipe is full, it waits for some space to be made or for no reader to be connected to the pipe
   iii. It may not write all the data was given, but it returns lenght of what it actually wrote

d. Behaviours b.ii and c.ii above can lead to your processes getting stuck on pipe operations. To avoid such situations, make sure you close each pipe end as soon as it is not needed anymore.

e. It is inherited from parent to child, this being the only mechanism of sharing it. Hence, in order for two processes to communicate through a pipe, one must inherit it from the other, or both must inherit it from a common ancestor.

f. Here is the re-implementation of the program above, with inter-process communication using pipes.

```
 1 int main(int argc, char**argv) {
 2     int a[4] = {1, 2, 3, 4};
 3     int pid, p[2];
 4
 5     pipe(p);
 6     pid = fork();
 7     if(pid == 0) {
 8         close(p[0]);
 9         a[2] += a[3];
10         write(p[1], &a[2], sizeof(int));
11         close(p[1]);
12         exit(0);
13     }
14     close(p[1]);
15     a[0] += a[1];
16     read(p[0], &a[2], sizeof(int));
17     close(p[0]);
18     wait(0);
19     a[0] += a[2];
20     printf("%d\n", a[0]);
21     return 0;
22 }
```

g. What would happen if we move line 14 between lines 5 and 6? Child inherits a closed pipe end, named p[1].
h. Advice : use a separate pipe for each direction of communication.
i. Attention : do not confuse pipe with its ends; there are always two ends to a pipe.

Example
Write a C program that runs a bash command received as a command line argument and times its execution.

```
/* Create a C program that runs a bash command received as a command line argument and prints
its execution time.
 * Ex:
 * ./exe grep -E -c "^[a-z]{4}[0-9]{4}" /etc/passwd
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/wait.h>
int main(int argc, char *argv[]) {
    struct timeval tv1, tv2;
    if (argc < 2) {
        printf("Please provide at least one argument.\n");
        exit(1);
    }
    gettimeofday(&tv1, NULL);
    int f = fork();
    if(-1 == f) {
        perror("Error on fork: ");
        exit(1);
    } else if (0 == f) {
        if( -1 == execvp(argv[1], argv + 1)) {
            perror("Error running the given command: ");
            exit(1);
        }
    } else {
        wait(0);
        gettimeofday(&tv2, NULL);
        printf("Total time = %f seconds\n", (double)(tv2.tv_usec - tv1.tv_usec) / 1000000 +
(double) (tv2.tv_sec - tv1.tv_sec));
    }
    return 0;
}
```

Unix Fifo Communication

1. What if we want to communicate between processes that do not have a common ancestor written by us? Since pipe are only transmitted through inheritance, we need another mechanism.
2. FIFOs are very similar in functionality to pipes and can be used to communicate between any processes.
3. The differences of using FIFO versus pipes are :
   a. FIFOs are files on disk, and consequently they have a unique system wide ID (the file path) that can be used by processes to address them.
   b. Pipe creation also opens them, but FIFOs need to be created and open explicitly
   c. Pipes are destroyed when all their ends closed, FIFOs need to be deleted explicitly
   d. FIFOs may live beyond the processes that use them, and if not properly handled, may also keep data during executions, thus potentially creating problems
4. How can we create FIFOs
   a. Using the command **mkfifo**
      i. Example : **mkfifo myfifo**
   b. Using C instruction **int mkfifo(const char *pathname, mode_t mode)**
      i. Example : **mkfifo("myfifo", 0600)**
5. FIFO acess control
   a. Having an unexpected process read or write from you in FIFO, will result to data being removed or added to the FIFO in ways your implementation does not expect
   b. For the beginning, it is a good policy to create FIFOs somewhere in your home directory, and five permissions only to yourself (i.e. 600)
6. FIFO removal can be done either with the **rm** command or the **int unlink(const char *path)** C instruction
7. Create FIFOs :
   a. Using the command **mkfifo w2r**

```c
// 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;
}
```

```c
// 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. Remove the FIFOs at the end using command **rm w2r**
8. Special **open** behavior for FIFO, when a process opens a FIFO with :
   - O_WRONLY → the open() blocks until another process opens the FIFO with O_RDONLY
   - O_RDONLY → the open() blocks until another process opens the FIFO with O_WRONLY

   It can lead to **deadlock** if two processes open two FIFOs in opposite order. For example :
   - FIFO1 = a_to_b
   - FIFO2 = b_to_a

   Program A:
   open("a_to_b", O_WRONLY); // waits for reader
   open("b_to_a", O_RDONLY); // never reached if blocked
   Program B:
   open("b_to_a", O_WRONLY); // waits for reader
   open("a_to_b", O_RDONLY); // never reached if blocked
   Both processes are waiting for the other to open the complementary side of FIFO. Since neither does, both are stuck - deadlock.