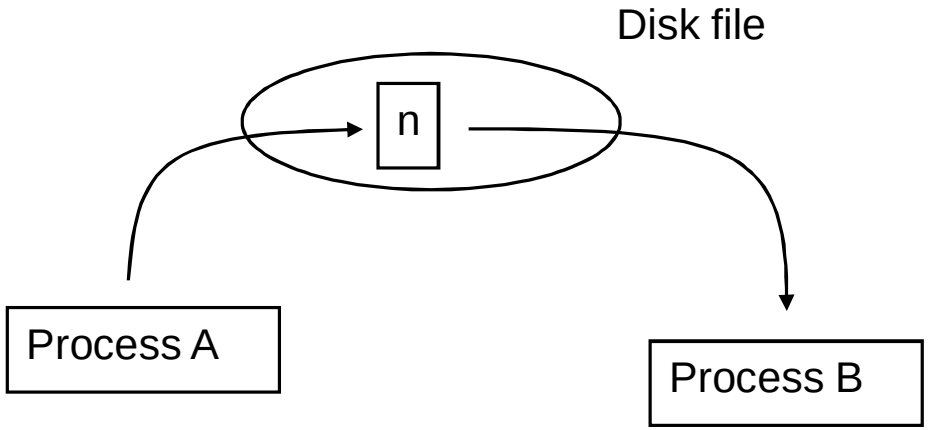
Real-Time Programming
TI00AA55

Lecture 9 - 25.03.2015

Jarkko.Vuori@metropolia.fi

Inter process communication via file

- IPC is an acronym for the term Inter Process Communication
- A file on the disk can always be used to pass information between processes
  - But then the synchronization is usually a problem
- Example:
  - Process A writes counter values to the disk file containing one integer and process B reads this file
  - The requirement is that process B can read each of the value process A has produced and that process B reads each value only once
  - Let's study the problems encountered when trying to meet the requirement



- Problems here are:
  1. Process A can write several values before B is scheduled to run
     - Process B misses values
  2. Process B reads same value many times
- Pipe is a simple solution to this problem

Inter process communication with pipes

- Pipe is an old and simple tool for inter process communication
- A process can send data to another process via a pipe
- The synchronization is a built-in feature of pipes
- If we use pipe in the problem on the previous page, the things happened in the following way:
  - Process A writes each counter value to the pipe
  - Process B reads counter values from the pipe
  - If pipe is empty meaning that no counter value is ready, process B blocks on the read operation and waits
  - If pipe is full, process A blocks on the write operation and waits
- The restrictions or disadvantages of the pipe are:
  1. In one pipe data can be moved to one direction only (you can of course choose whether it goes from process A to process B or vice versa)
  2. It is possible to establish a pipe only between “relatives” (between parent and child or between children of same parent)
- The latter restriction can be avoided with named pipes (FIFO)
- Most often pipes are used in the communication between parent and child

Creating and using a pipe

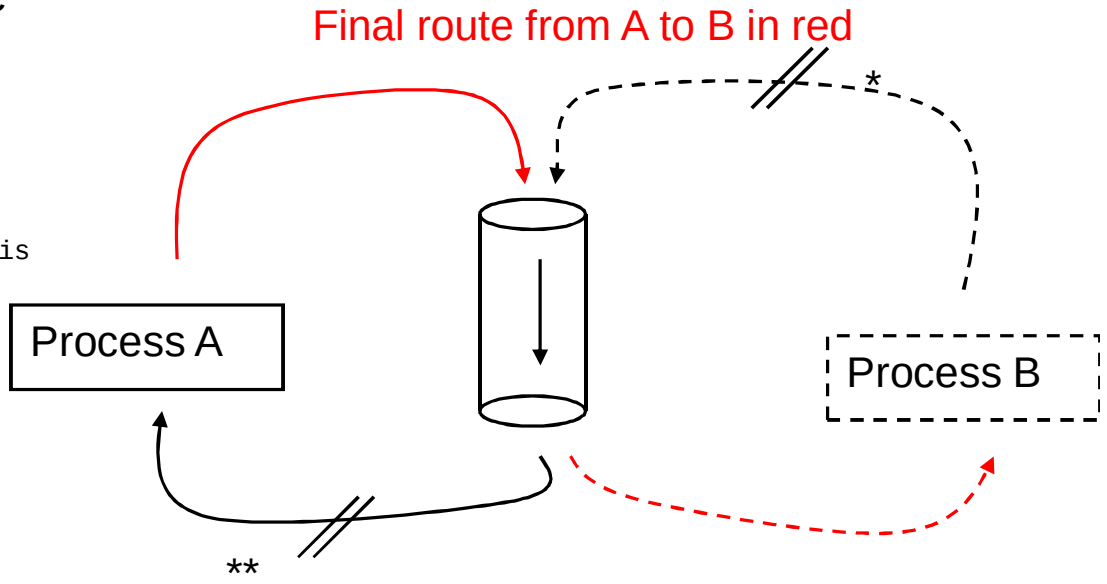
- Function pipe creates a pipe
```
<unistd.h>
int pipe(int fd[ ]);  or
int pipe(int* fd);
```
  - returns 0, if OK, and -1 if error
- The parameter of this function is an array of two file descriptors
  - When the function returns, this array contains two opened file descriptors
- Array element 0 contains file descriptor for reading from the pipe (read end) and array element 1 contains file descriptor for writing to the pipe (write end)

- All data written to the pipe (to the file descriptor fd[1]) can be read from the other end of the pipe (from the file descriptor fd[0])
- The file type of pipe is FIFO
  - If you call function fstat for “pipe descriptors” and test the field st_mode of struct stat with macro S_ISFIFO, the result is true
- Normally writing process closes the descriptor fd[0] and reading process closes the descriptor fd[1], so that each process has access only to one end of the pipe

Example and illustration of pipe creation

- The following short example illustrates how to create a pipe from the parent process to the child process

```
int fd_arr[2];
  ...
pipe(fd_arr); // Solid things exist after this
pid = fork(); // Also dotted things exist after this
if (pid == 0) {
  close (fd_arr[1]);  // see * in the figure
  read(fd_arr[0], &chr, 1);
    ...
}
if (pid > 0) {
  close (fd_arr[0]) ;  // see ** in the figure
  write(fd_arr[1], "A", 1);
    ...
}
```



A complete pipe example program

```
// This is simple pipe example
// Parent writes to the pipe. Child reads from the pipe
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(void) {
    pid_t pid;
    int fd_arr[2], i, n;
    char chr;
    pipe(fd_arr);
    pid = fork();
    if (pid == 0) { // This is child
      close(fd_arr[1]); // Child closes it's write end of the pipe
      // Reading from the pipe
      for(i = 0; i < 10; i++) {
           read(fd_arr[0], &chr, 1);
           printf("%c\n", chr);
      }
      close(fd_arr[0]);
      exit(0);
    }
    // Parent continues from here
    close (fd_arr[0]); // Parent closes it's read end of the pipe
    // writing to the pipe
    for(i = 0; i < 10 ; i++) {
      chr = 'a' + i;
      write(fd_arr[1], &chr, 1);
      sleep(1);
    }
    close(fd_arr[1]);
}
```

What happens, if we change 10 to 20 only here. See question 1 on page 8

What happens, if we change 10 to 20 only here. See question 2 on page 8

Pipe synchronization

- By default, the pipe descriptors are in blocking mode
- The pipe descriptor can be set to non-blocking mode (using the system call `fcntl`)
- In blocked mode the read operation blocks if pipe is empty
- The write operation blocks if there is no room enough in the pipe to write all the data indicated in the write call

- More about synchronization:
  - If the writing process is slower than the reading process → reading process waits for data (is blocked) for some time
  - If the reading process is slower than the writing process → writing process waits (is blocked) until there is room enough in the pipe to write the data
- In Linux 2.6.11 and later, the size of pipe is 65536 bytes
- Constant PIPE_BUF is defined in limits.h. It is guaranteed, that writing process is atomic as long as the number of bytes does not exceed the limit PIPE_BUF
  - The value is 4096 in Edunix

Special situations

- Different kind of special situations:
  - Trying to read from an empty pipe, when the write end is closed → read returns with a return value 0 (eof)
  - Trying to write to a pipe when the read end is closed → the signal SIGPIPE is sent to the writing process. If the signal handler returns, the function write returns with errno EPIPE
- Lets return to the example on the page 6. The reading process “knows” how many bytes to read. In real situations this is not usually true. Often for example, the reading terminates, when end of file has been received. Lets consider the following situations and answer the questions.
  - What happens if we change the number of steps in the reading process (in the child process) from 10 to 20? The sending process still sends only 10 characters
  - What happens if we change the number of steps in the writing (in the parent process) from 10 to 20? The reading process still reads only 10 characters
  - What happens in situation 1, if we comments out the statement close (fd_arr[1]); in the child? What happens in situation 2, if we comments out the statement close (fd_arr[0]); in the parent?

Pipe in non-blocking mode

- The pipe descriptor can be set to non-blocking mode (using the system call fcntl)
- In non-blocking mode the behaviour with write system call is as follows:
  1. If number of bytes is $\leq$ PIPE_BUF
     - If there is room to write all n bytes to the pipe, then write succeeds immediately and all bytes are written
     - If all n bytes cannot be written immediately, write returns immediately with the return value -1 and with errno set to EAGAIN.
  2. If number of bytes is > PIPE_BUF
     - If the pipe is full (there is no room at all), write returns immediately with errno set to EAGAIN
     - Otherwise write writes as many bytes as it is possible to write without blocking. So it is important to check how many bytes really were written. Data can be written only partially
- Remark. Another thing that can happen, when n > PIPE_BUF, even if pipe is in blocking mode, is that the bytes sent with one write are not necessarily contiguously in the pipe. If other processes also write to the same pipe, the data from one process may be interleaved with data from another process
- In non-blocking mode read always returns immediately. If pipe is empty, return value is -1 and errno EAGAIN

Recap of descriptor features

- File descriptors are process related and they are stored in the file descriptor table of the process that kernel maintains
- When a file is opened, kernel creates an entry in the file table in addition to the entry in the file descriptor table
- When a process forks a child, open file descriptors are **duplicated** for the child process
  - This means that child has it's own file descriptor table with the same descriptor numbers **pointing to the same file table entries** than corresponding descriptors in the parent
- A process can also duplicate a file descriptor for itself
  - Functions dup and dup2 duplicate an entry in the file descriptor table
  - Then two file descriptor table entries point to the same entry in the file table
- Entry in the file table is deleted only when the last descriptor that points to that entry is closed
- Exec functions change the running program in the process, but process remains the same. This means for example, that the file descriptor table of the process is not modified in exec. Open file descriptors are still there, provided that file descriptor flag close-on-exec is not set on
- Often we want to change the program at one or both ends of the pipe. The new program then needs to know the pipe descriptor so that it can access file descriptor table correctly

How to pass file descriptor to a new program

- We have earlier learned how to pass a file descriptor as a command line parameter to the new program
- File descriptor needs to be converted to the string first
- The exec command would look as follows

```
execl("./pipe_end", "pipe_end.exe",
fd_string, (char*) 0);
```

- There is another option. We can “connect” a descriptor to another known descriptor using dup2 function
  - Most often a descriptor is duplicated to the well known descriptor like STDIN_FILENO or STDOUT_FILENO
- Example. We now want our child process to write to the pipe and run the new program pipewrite in the child. The child program **writes to the standard output** (to the display basically), but everything goes to the pipe. The code that forks a child and execs a new program in the child looks as follows:

```
pipe(fd_arr);
pid = fork();
if (pid == 0) { //This is a child
  close(fd_arr[0]); //Close read end
  if (fd_arr[1] != STDOUT_FILENO) {
    dup2(fd_arr[1], STDOUT_FILENO);
    close(fd_arr[1]); //Close original pipe end
  }
  execl(("./pipewrite", "pipewrite", (char*)0);
}
```

Recalling close-on-exec flag

- As said before, the open file descriptors of the child are still open and usable after the child has called exec
  - This can be prevented from happening with the file descriptor flag close-on-exec. (This flag is a file descriptor flag, not a file flag. File descriptor flags are stored in file descriptor table. The file flags on the other hand are stored in file table entries.)
- If close-on-exec bit is set, the file descriptor is closed, when the process calls exec
  - By default this bit is off
- The file descriptor flag close-on-exec can be turned on in the following way:

```
int fd_flags;
fd_flags = fcntl(fd, F_GETFD);
fd_flags |= FD_CLOEXEC;
fcntl(fd, F_SETFD, fd_flags);
```

- One small additional example of descriptors:

```
int fd1, fd2, fd3;
fd1 = open("data.dat", O_RDONLY);
fd2 = open("data.dat", O_WRONLY);
```

- In this example two entries are created in the file descriptor table as well as in the file table
- If we add the following line to the program

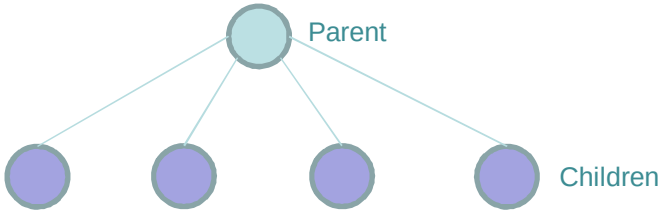
```
fd3 = dup(fd2);
```

  we have three entries in the file descriptor table but still only two entries in the file table

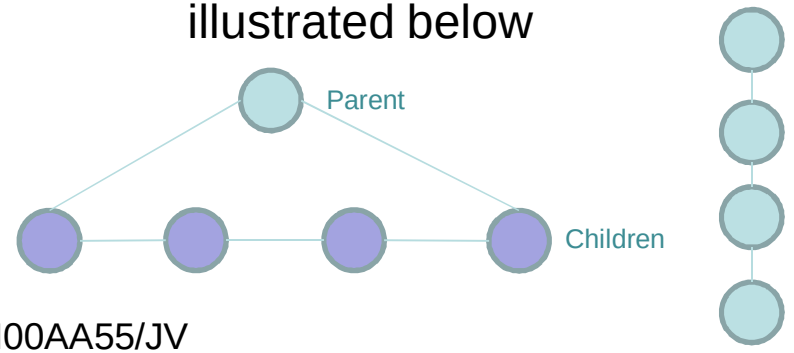
Different ways to use pipe

- There are many design patterns where pipes are used
- These are some of them:
  - The parent process has a connection to a child or to several children
  - Different ”production pipelines” or conveyors
  - Barrier
  - Pipe as a synchronization tool (semaphor)
  - Piping in the shell

- The principle of the situation where the parent has a connection to the children can be seen below
  - The connection can be created from the parent to the children or vice versa, or even to both directions if there are two pipes between the parent and a child



- Two different pipeline solutions are illustrated below



Barrier

This is how parent process can make sure that all children are created (are existing), before any of them start to work

```
// This program demonstrates a barrier
#define N  100
int main(void) {
  pid_t pid;
  int pipe_fd[2];
  int i;
  char chr;
  pipe(pipe_fd);
  for (i = 0; i < N; i++) {
    pid = fork();
    if (pid == 0) { // this is child
      read(pipe_fd[0], &chr, 1);  // each child waits here until all are ready
      printf("Child is running\n");
      sleep(i+1);
      exit(0);
    } else
      printf("Child created\n");
  }
  for (i = 0; i < N; i++)      // After creating all children
    write(pipe_fd[1], "A", 1); // the parent gives children
  …                            // a permission to continue
}
```

PIPE as a synchronization tool 1

- Pipe is actually a method to send data from one process to an another in a synchronized way
- It is possible however to use it purely for synchronization (barrier actually also was that kind of example)
- We have earlier used signals (and sigsuspend or sigwait) for synchronization
- Now we again solve the same kind of problem with two pipes
  - One pipe leads from the parent to the child, and an another from the child to the parent

- The basic idea is as follows:
  1. When we want the parent process to wait for the child, we put the parent to read from the pipe from the child
  2. When we want the child process to tell the parent that it can continue, we put the child process to write to the pipe from the child
  3. When we want the child process to wait for the parent, we put the child to read from the pipe from the parent
  4. When we want the parent process to tell the child that it can continue, we put the parent process to write to the pipe from the parent
- The following four functions can be written (see simplified implementations on the next page):

```
void wait_child(); void
tell_parent_to_continue();
void wait_parent(); void
tell_child_to_continue();
```

- The fifth function

```
void initialize_sync(void);
```

creates the two pipes

PIPE as a synchronization tool 2

synchronize.h

```
void initialize_sync(void);
void wait_child();
void wait_parent();
void tell_parent_to_continue();
void tell_child_to_continue();
```

synchronize.c

```
static int pipe_from_child[2], pipe_from_parent[2];
static char c = 'c', p = 'p';
void initialize_sync(void) {
    pipe(pipe_from_child);
    pipe(pipe_from_parent);
}
void wait_child() {
    char chr;
    read(pipe_from_child[0], &chr, 1);
}
void wait_parent() {
    char chr;
    read(pipe_from_parent[0], &chr, 1);
}
void tell_parent_to_continue() {
    write(pipe_from_child[1], &c, 1);
}
void tell_child_to_continue() {
    write(pipe_from_parent[1], &p, 1);
}
```

PIPE as a synchronization tool 3

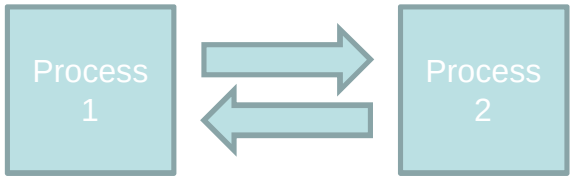
```
// This is an example that shows how to synchronize
// the parent process and the child process using
// functions represented on the previous page
int main(void) {
  pid_t pid;

  initialize_sync();
  pid = fork();
  if (pid == 0) { // this is child
    while (1) {
      write(STDOUT_FILENO, "1", 1);
      tell_parent_to_continue();
      wait_parent();
    }
  }
  if (pid > 0) {
    while (1) {
      wait_child();
      write(STDOUT_FILENO, "2", 1);
      tell_child_to_continue();
    }
  }
}
```

Piping in the shell

- Everybody has used pipe symbols in the command line so that the output of the first program is sent as an input to the second program
  - E.g. `who | grep ken`
- Shell also uses the techniques we have just learned
- The example program shell_own_3_pipe.c demonstrates how shell handles the piping commands like `progr1 | progr2`

More about application models for pipes

- There are many applications for pipes as we have seen
- We still return to two cases, that actually are special cases of those already handled
- Pipes can be used to construct a so called **parallel processing model**. The principle of parallel processing model is illustrated in the figure right
  - Process 1 uses the services of process 2 for processing data
  - Two pipes are used to construct this kind of model
  - One for sending the data to the process 2 and another to get the results from that process



- Another application model is a **pipe line model**. The basic idea is that one process makes pre-processing for the data and sends it to the next process for further processing. This second process makes further refinements for the data and sends it to next process in the pipe line and so on
- The principle of the pipe line system is illustrated on the next page

Using library functions in pipe applications

- The following figure illustrates the pipe line consisting of three processes

- We have already learned how to implement this kind of system using functions pipe, fork and exec. The pipe line model can also be implemented using library functions. C standard **library functions** `popen` and `pclose` support the constructions of the pipeline

- The construction of the pipe line above with these functions looks as follows:
  Process 1 (Program preprocess.c)

```
int main(void) {
  Tdata_orig data_orig; Tdata1 data1;
  FILE* to_middleproc;
  to_middleproc =
  popen("middleprocess.exe", "w");
  while ( ... ) {
    // get original data and preprocess it
   fwrite(&data1, sizeof(data1), 1,
          to_middleproc);
      ...
  }

  pclose(to_middleproc);
}
```

Pipe line with library functions (cont.)

Process 2 (Program middleprocess.c)

```
int main(void) {
  Tdata1 data1; Tdata2 data2:
  FILE* to_finalproc;
  to_finalproc = popen("finalprocess.exe", "w");
  while ( ... ) {
    // read data from the process 1
    fread(&data1, sizeof(data1), 1, stdin);
    // process the data1
    // result is stored in data2
    // send the result to the process 3 to be finalized
    fwrite(&data2, sizeof(data2),1, to_finalproc);
          ...
  }

  pclose(to_finalproc);
}
```

Process 3 (Program finalprocess.c)

```
int main(void) {
  Tdata2 data2; Tdata_out data_out;
  while ( ...) {
    // read data from the process 2
    fread(&data2, sizeof(data2), 1, stdin);
    // process the data
    // provide final product data_out

  }
}
```

Functions popen and pclose

- In the previous example we saw a function call like
  to_middleproc = popen(”middleprocess.exe”, ”w”);
  in the process 1
- The function `popen` does in this case the following things:
  1. Creates a pipe (pipe)
  2. Forks a child process
  3. In addition to that, the **parent** process:
     - Closes the read end file descriptor of the pipe
  4. In addition to that the **child** process:
     - Closes the write end file descriptor of the pipe
     - Duplicates the read end file descriptor to the standard input STDIN_FILENO
     - Closes the original read end file descriptor of the pipe
     - Calls exec to start to run the program middleprocess.exe in the child
- The function `pclose` in the process 1 in this case does the following things:
  1. Waits for the child process
  2. Closes the write end of the pipe

FIFO basics and creation

- The disadvantage of pipe is that it can be used only between relative processes (between parent and child process or between sisters)
  - FIFO removes this limitation
  - Independent processes can communicate with FIFO
- Another name for FIFO is named pipe
  - FIFO behaves in very similar way than pipe. The only difference is in the way it is created
- FIFO has a ”public” file name in a directory structure that represents a FIFO
  - FIFO is a special file (file type FIFO)
  - File name represents a FIFO (operating system service to pass information between independent processes). It does not represent any real disk file
- FIFO can be created with the function

  ```
  int mkfifo(const char *pathname, mode_t mode);
  ```

  Function returns 0, if OK and -1 if not OK
- The parameter mode is like the third parameter in the file open call, when file is created (flag O_CREAT is set in the second parameter)
  - Parameter mode is used to express the access rights for the FIFO

Opening FIFO

- When FIFO has been created with the function `mkfifo`, it can be opened for writing or reading with the function open like ordinary files
  - Open returns file descriptor to the read end of the FIFO with the access mode O_RDONLY or to the write end of the FIFO with the access mode O_WRONLY
  - If access mode is O_RDWR, the same file descriptor represents both ends of the FIFO
- Notice that what ever process that knows the name of the FIFO can open it and start to use it
- File descriptors that have been returned by the open function behave like file descriptors in the file descriptor array after calling pipe function
- Usually one end of the FIFO is opened for reading and another end for writing
- If FIFO is opened in read only mode, the open call blocks if another end is not opened for writing. If FIFO is opened in write only mode, the open call blocks if another end is not opened for reading. Both of these statements are true if the open flag O_NONBLOCK is not set (this is default)
- One process can open a FIFO for reading and writing (can open both ends). In this case the open call does not block

More about FIFOs

- As said before, after opening the FIFO descriptors, FIFO behaves in the same way as the pipe. The following things are still valid
- The data moves only to one direction in the FIFO (as it did in the pipe)
- The synchronization works in similar way than with pipes
  - This means that read function blocks if the FIFO is empty and write function blocks if FIFO is full
- If the process tries to write to the write end of the FIFO and the read end of the FIFO is already closed (there is no reader any more) a signal SIGPIPE is generated
  - The default handler of this signal terminates the whole process
  - If we have a signal handler that returns (return) the write function returns with an error code EPIPE
- When the last writer has closed the write end of the FIFO and the reader reads from an empty FIFO, read function returns 0 (eof)
- It is guaranteed that the write operation to the FIFO is atomic if the number of bytes is not greater than constant PIPE_BUF
- Read operation from the FIFO is not necessarily atomic. Read can return before all bytes are read
- See examples fiforeader.c and fifowriter.c

Non-blocking use of FIFO

- The synchronization works inherently and transparently when FIFO is used in blocking mode
- It is possible however to set the FIFO to the non-blocking mode
- FIFO can be set to non-blocking mode, when it is opened
- Everything that is said on the page 8 is also true for non-blocking FIFO descriptors
- The O_NONBLOCK flag has effect also in the open calls of the FIFO (not only in read and write operations)
- If the flag O_NONBLOCK is set in the open call and access mode is O_RDONLY, open returns immediately **without any error** even if the FIFO is not yet opened for writing. If you read the FIFO after that, you will get eof, because there is no writer
- If the flag O_NONBLOCK is set in the open call and access mode is O_WRONLY, open function returns immediately **with an error ENXIO** (No such device or address), if another end is not yet opened for reading

Client/Server with FIFOs

- FIFO can be used to implement a client server like system, because many writers can write simultaneously (concurrently) to the FIFO so that it can be used as a “requests fifo” that client processes can use to request services from the server
  - Because FIFO can be accessed using the file name, what ever process that knows the FIFO name can use the server
- If the server needs to send some data to the clients, the solution is not self evident. We could try to use another common FIFO from the server to all the clients. In principle, the FIFO can have many readers but it is difficult to make sure that each reader gets what is really meant for it. In other words, it is difficult to synchronize the reading between processes
- One method to solve this problem is that clients send their process IDs to the server (as a part of the service request) and the server creates a separate FIFO for each client according some naming rule for giving a service or sending a reply to the client. One possible naming rule would be for example

  server_name.xxxx

  where xxxx is an process ID of the client process
- Sockets provide a better method to implement servers, when data is also sent from the server to the client