First part of topic 11

„Tools of InterProcess Communication
(file, signal, pipe, socket)"

A **process** is a series of activities that occur on a computer by starting an executable program, that is, an execution instance of the program. Implementation can take place on multiple threads. Processes can have different states (created, ready, running, waiting, terminated, etc.). Today's multi-programing operating systems can have more than a hundred processes present at a given time moment. Each has a unique identifier, the PID. The seemingly continuous operation of these processes is aided by the scheduler. Processes use resources exclusively (memory, files, devices, etc.). This requires mutual exclusion and process synchronization. Processes often need to exchange information with each other.

For the above reasons, **interprocess communication** (IPC) is essential. There are different ways to communicate. Most modern operating systems support:

- file
- signal
- socket
- pipe
- shared memory
- semaphore
- etc.

File

Processes can access elements of the file system stored in mass storage. In one approach to communication, one process can write the contents of a file and the other is then able to read the contents, thereby realizing the exchange of information. This provides indirect, two-way, asynchronous communication, enabling large messages. There are two different approaches to file management: **text-based** (or high-level/formatted) and **binary** (low-level) file management. In the former case, the strings stored in the file are converted to different types of values during the scan. (For example, the "-12345.67890" Latin-1 (ISO 8859-1) encoded 12-byte string that is part of the file can be stored in a double-precision floating-point representation (IEEE 754/1985) 8-byte variable after conversion.) For binary files, the same bit sequence represents the value in both the file and memory. (For example, the string "Imre" can mean an integer – the 1701997897 – with signed fixed point representation stored in the four-byte using little-endian byte order.)

Whether it's text or binary file management, the following basic operations can be performed:

- opening
- writing
- reading
- positioning the stream

- closing

Files must be opened before use. During the **opening**, you decide which file you are going to work with, how you want to use it (write only, read only, read and write, append). When the file is opened, a file descriptor is created, so that we can refer to the logical file later. For text files the `fopen` function, for binary files the `open` system call can be used in C programming language.

By **writing**, you can store values in the file, and by **reading**, the data stored in the file is copied into the memory. For example, the `fprintf/fscanf` functions can be used to write/read text files, and the `write/read` system calls can be used for binary files.

Files always have a working position (a kind of imaginary cursor) that means where the next read/write operation will begin to take effect. The default value of the position is determined at opening, and then automatically changes according to the number of bytes written/read during each writing/reading operation. It is also possible to change the stream position manually using the `fseek/lseek` functions.

After completing the desired actions, the opened files must be closed. For text files using the `fclose` function, for binaries using the `close` system call. This saves the changes, changes the attributes of the file, and makes the file content available to other programs.

Signal

It is basically used to synchronize processes, which can also be interpreted as a kind of direct, one-way communication. A signal is a kind of warning (without information content) that one process can send to another. There are several types of signals to indicate the occurrence of different events. For example: `SIGKILL, SIGTERM, SIGINT, SIGALRM, SIGCHLD, SIGSTOP, SIGCONT, SIGUSR1, SIGPIPE,` etc. These fall into two categories, there are **catchable** and **non-catchable** signals. When the latter arrive, a well-defined action will be performed. In the case of the former, you can prepare for the arrival of the signal, the programmer can determine what to do in case the signal arrives (if it arrives at all). The activity to be performed must be defined as a **signal handler** procedure. To associate the signal with the given signal handler, we use the `signal` function for the C language. After the above-mentioned binding has taken place, the execution of the program continues as usual, but later, if the given signal arrives at any time, the normal execution is suspended, the signal handler procedure runs through an automatic hidden call, and then the execution of the program continues with the following instruction. (The logic is similar to handling a hardware interrupt.)

A process can send a signal to another process. This is what the `kill` function is for in C. It can be used to send any signal to a process having a given PID value. If the destination process is prepared for the arrival of the given signal, then synchronization takes place, i.e., one process can know about the other where it “goes” with the execution. (When the signaling happens both processes are in a well-defined point of their executions.) Using the `pause` function, a process is put on hold until it receives a signal. Then the given signal handler runs and then the normal execution of the program continues, so that one process can “wait” for the other.

Pipe

It enables FIFO-type, two-way, asynchronous communication by implementing an imaginary pipeline between the two processes. What a process puts into the pipe goes to the other, who can get it sequentially. The pipeline can be anonymous or named.

A simple special case is when the default output of a process is redirected in a command line environment to the default input of another process, implementing one-way communication using the operator marked with a `'|'` (vertical line, pipe) character. For example, on Linux, the "`less tmp.txt`" command displays the contents of a given text file on the screen (standard output), while the "`wc -c`" command specifies the number of characters in the typed text (standard input). In this case, the complex "`less tmp.txt | wc -c`" command determines the number of characters stored in the text file via the pipeline, because the file content does not appear on the output but serves as input for the other process.

In the C language one can create a pipe with the `pipe` function. Its parameter is the address of an array of two integers, where after the invocation, we can find two low-level file descriptors. The second element of the array identifies a special binary file for writing, while the first array element is the identifier of a readable file. Programmer can use the `write` and `read` system calls to send and receive via this sequential dataflow. After a `fork` operation the two ends of the pipe can connect the child and the parent processes, implementing one directional inter-process communication.

Socket

The socket is the endpoint of TCP/IP-based communication between two processes. These endpoints can also belong to two processes on the same computer or on different computers. There are several types of sockets. One is the most commonly used "datagram" type to implement connectionless communication using the UDP transport layer protocol, the other is the "stream" type, which is connection-oriented due to the TCP protocol used. The procession of client-server architecture communication is different for the two types, although there are necessary steps. Before communication, the socket must be created (`socket`), then we can send (`send, sendto, write`) and receive (`recv, recvfrom, read`) messages and at the end it must be closed (`close`). On the server side, the socket must always be assigned to a network addresses (IP address and port number). In the case of stream communication, a connection must be established using the `listen` and `accept` functions on the server side and the `connect` function on the client side at the appropriate time after the sockets have been created, but before the actual message is sent. These functions use several special data structures and types (for example: `in_addr, sockaddr_in, sockaddr`, etc.).

Comment:

The ability to use the programming tools associated with the topic may be required. Wider familiarity with the subject is not a disadvantage. This is just a short summary.

Related subjects:

- *System programming*
- *Computer architectures*
- *Operating systems*

Further suggested readings:

- Niel Matthew, Richard Stones: *Beginning Linux Programming* (Wiley, 2004) Chapter 3, 11, 13 and 15.