Interprocess communication

Interprocess communication

Classic tools:

- Environment variables
- Signals
- Pipes
  - Named
  - Unnamed
- Sockets

Environment variables

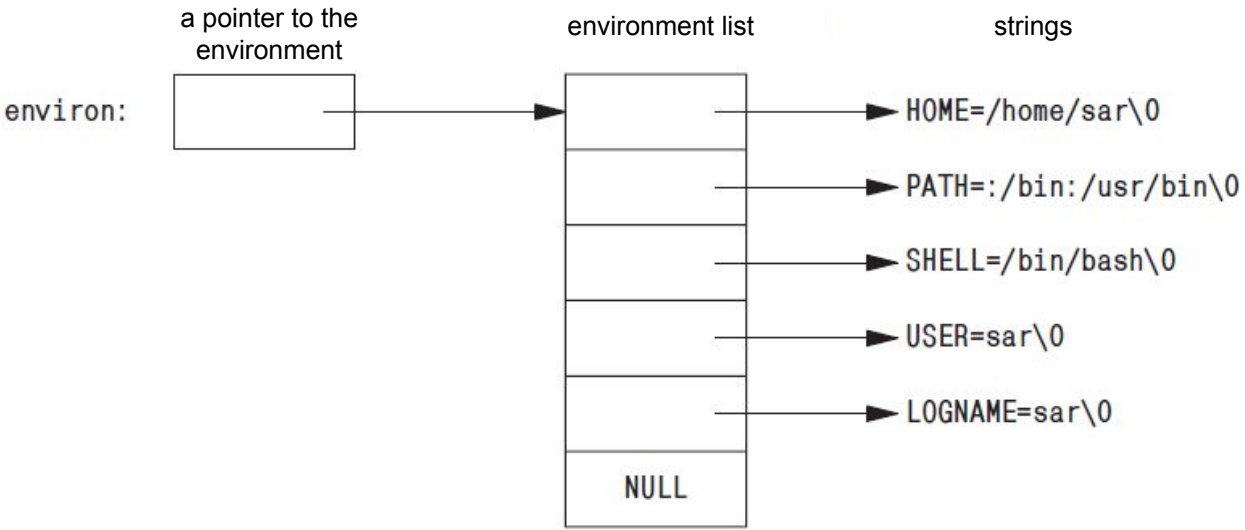
Environment variable - a text variable of the operating system that stores any information - for example, data on system settings

An environment variable is a dynamic-named value that can affect the way running processes will behave on a computer

They are part of the environment in which a process runs. For example, a running process can query the value of the TEMP environment variable to discover a suitable location to store temporary files

List of environment variables

extern char **environ;



Example

Displays all environment variables of the current process

Example environ.c

System calls

#include <stdlib.h>

char ***getenv**(const char *name);

int **putenv**(char *str);

int **setenv**(const char *name, const char *value, int rewrite);

int **unsetenv**(const char *name);

Environment variable

- Unidirectional action
- You cannot change the environment of an already running process from the outside
- You can pass only small amount of text data

They are mainly used to set the conditions for running the program

Signals

Signals - software interrupts providing asynchronous event processing.

Possible actions for the event:

- Ignore the signal (except SIGKILL and SIGSTOP)
- Intercept the signal and process it
- Default action

Signals

SIGALRM (14)	The process can use a special abort system call to set the time after which it needs to send a signal. After a specified period of time, the operating system will deliver a SIGALRM signal to the process. Typically, this technique is used to set timeouts. If the process has not registered a handler for this signal, the default handler terminates the process.
SIGCHLD	The signal is sent to the parent process when its child process is completed. By default, the signal is ignored.
SIGCONT	Signal to continue program execution after stopping. There is no default handler.
SIGFPE (8)	Error signal in floating-point calculations is sent by the operating system in case of incorrect execution of the program. The default handler terminates the process.
SIGHUP (1)	A signal to close the terminal to which this process is linked. It is usually sent by the operating system to all processes running from the command line when the user logs off. The default handler terminates the process.

Signals

SIGKILL (9)	Signal to abort the process. On this signal, the process ends immediately — without releasing resources. This signal cannot be intercepted, blocked or overridden by the process itself, and the standard operating system handler is always used. This signal is used to guarantee the completion of the process.
SIGPIPE (13)	A signal is sent to a process that tries to send data to a pipe that is closed on the opposite side. This situation may occur if one of the interacting processes has been terminated abnormally. The default handler terminates the process.
SIGSEGV (11)	The signal is sent to the process by the operating system, if an incorrect memory operation was performed (access to a non-existent or protected address). The default handler terminates the process.
SIGSTOP	The signal to suspend the process. This signal cannot be intercepted, blocked, or overridden. It is used to guarantee the suspension of the process with full preservation of its state and the ability to resume.

Signals

SIGTERM (15)	A process shutdown signal is typically used to complete the process correctly.
SIGUSR1, SIGUSR2	"Custom" signals - can be used by processes for all kinds of notifications. The default handler terminates the process.

Signals

Displays a list of system-specific signals:

kill -l

System call

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal (int signo, sighandler_t
handler);
```

signal()

handler:

- void my_handler (int signo) function);
- SIG_DFL - default action
- SIG_IGN - ignore signal

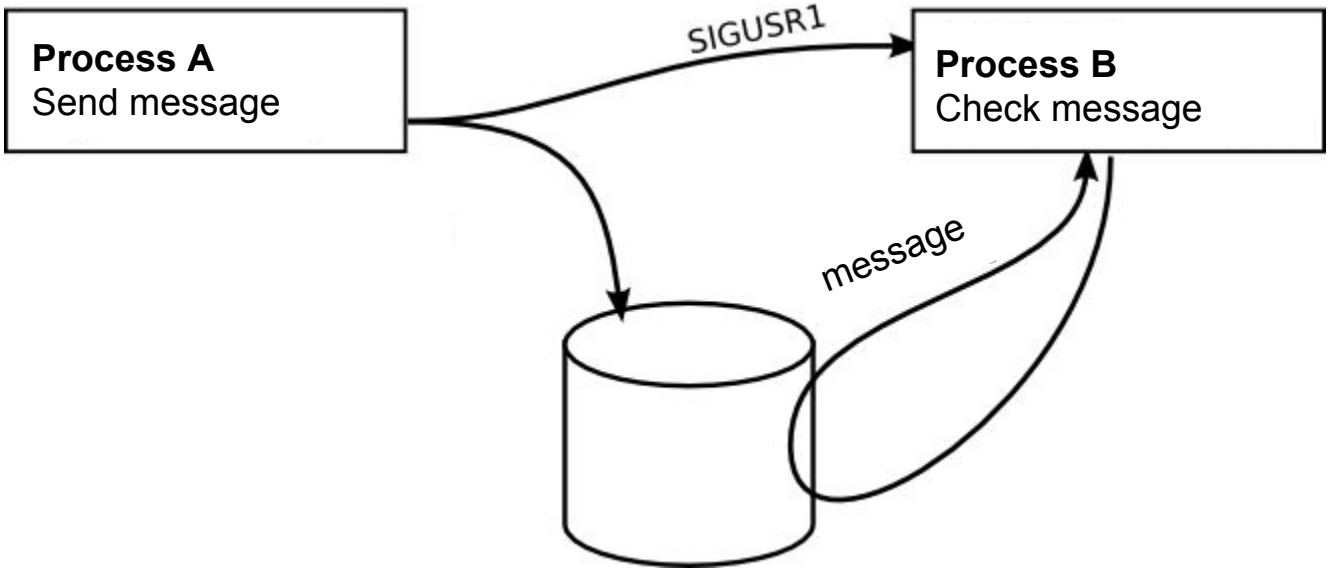
Example

The program is not closing at Control+C

signal.c

Signals

Using signals in interprocess exchange



Signals

Different behavior on system calls

fork and **exec**

What happens with signal handler when we invoke **fork()** system call?

What happens with signal handler when we invoke **exec()** system call?

Fork

The child process inherits the signal actions from its parent

Pending signals are not inherited

Exec

All signals operate in the default mode

Signals waiting to be processed are inherited

Sending a signal

```
#include <sys/types.h>
#include <signal.h>

int kill (pid_t pid, int signo);
```

Returns 0 if successful

Errors

-1 in case of error and errno setting:

EINVAL — the signal indicated by signo is not valid;

EPERM — the calling process does not have sufficient privileges to send a signal to any requested process;

ESRCH — the process does not exist, or is the process is a zombie.

The definition of access rights

```
int ret;
ret = kill (1722, 0);
if (ret != 0)
    ; /* the right of access is missing */
else
    ; /* access granted */
```

Unnamed pipes

Pipe - data flow between two or more processes that has an interface similar to reading or writing to a file

Pipes have two limitations:

1. Historically, they are simplex (that is, data can only be transmitted in one direction).

2. Unnamed pipes can only be used to communicate between processes that have a common ancestor. Typically, a pipe is created by a parent process, which then calls the fork function, after which the channel can be used to communicate between the parent and child processes.

Working with pipes

- One process writes, the second reads as from the file
- There is no limit to the size of data transferred
- There is a limit to the size of the buffer

Working as a file

Use as a descriptor:

open - is not necessary as the descriptor is created by a call to **pipe**

write - blocks execution if there is no buffer space

read - destroys data when reading, does not work lseek

When writing without readers, the process receives a **SIGPIPE** signal

Creation

```
#include <unistd.h>

int pipe(int filedes[2]);
```

Returns 0 on success, -1 on error

Example

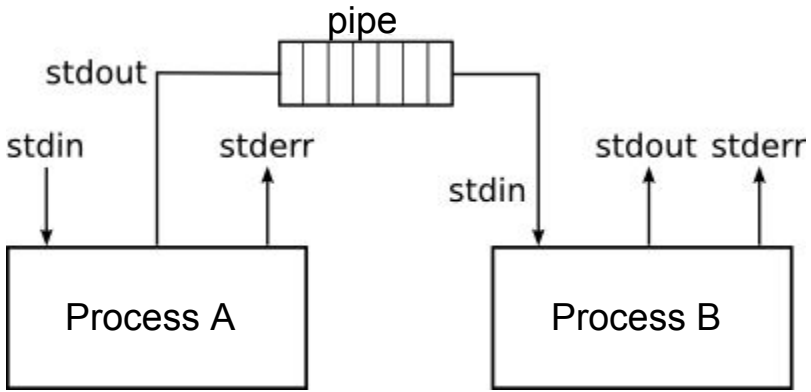
Passing data from the parent process to the child through a pipe

pipe.c

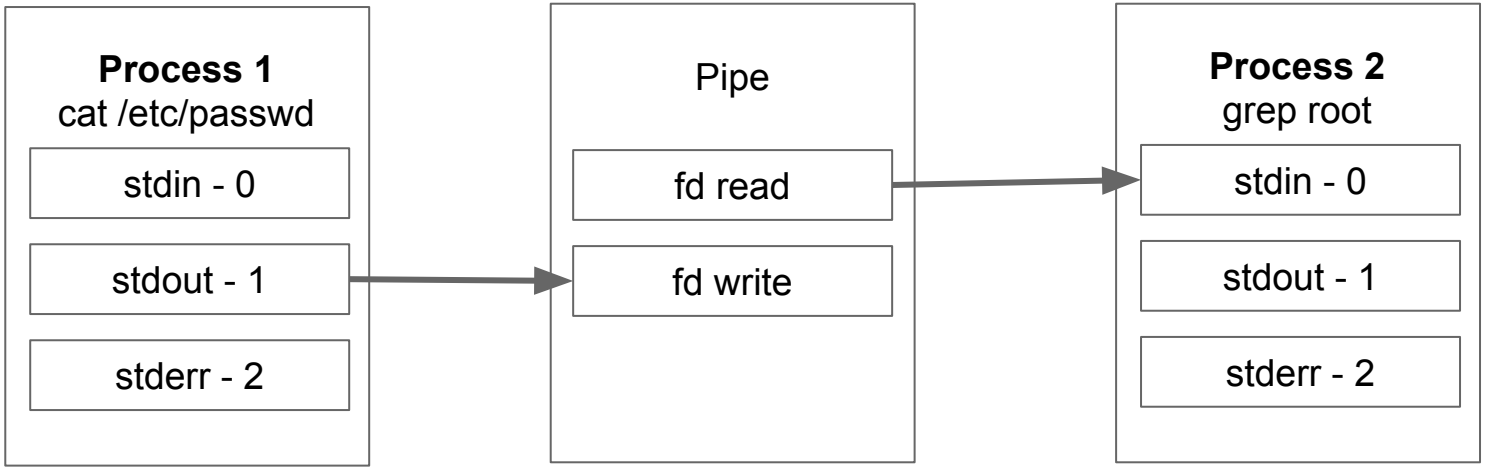
Unnamed pipes

Implementing the conveyor on the command line:

cat /etc/passwd | grep root



Conveyor implementation



Example **conveer.c**

Exercise №2b

Parallel computation of PI based on the following series:

$$Pi=4 - 4/3 + 4/5 - 4/7 + \ldots + (((-1)^{(n+1)})*4)/(2*n-1)$$

Divide the range by the number of processes P (N/P). Create P processes. Each process calculates its part of the sum and displays it on the screen

Named pipe (FIFO)

FIFO - a special type of file

Provides the ability to exchange data with unrelated processes

FIFO

#include <sys/stat.h>

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

Returns 0 on success, -1 on error

Example: client-server