Linux System Programming Part 5 - Interprocess Communication (IPC)

IBA Bulgaria
2018

IPC Methods

File	A record stored on disk, or a record synthesized on demand by a file server, which can be accessed by multiple processes.
Signal	A system message sent from one process to another, not usually used to transfer data but instead used to remotely command the partnered process.
Socket	Data sent over a network interface, either to a different process on the same computer or to another computer on the network. Stream-oriented (TCP; data written through a socket requires formatting to preserve message boundaries) or more rarely message-oriented (UDP, SCTP).
Unix domain socket	Similar to an internet socket but all communication occurs within the kernel. Domain sockets use the file system as their address space. Processes reference a domain socket as an inode, and multiple processes can communicate with one socket.

IPC Methods (2)

Message queue	A data stream similar to a socket, but which usually preserves message boundaries. Typically implemented by the operating system, they allow multiple processes to read and write to the message queue without being directly connected to each other.
Pipe	A unidirectional data channel. Data written to the write end of the pipe is buffered by the operating system until it is read from the read end of the pipe. Two-way data streams between processes can be achieved by creating two pipes utilizing standard input and output.
Named pipe	A pipe implemented through a file on the file system instead of standard input and output. Multiple processes can read and write to the file as a buffer for IPC data.
Shared memory	Multiple processes are given access to the same block of memory which creates a shared buffer for the processes to communicate with each other.

Signals

- **Signals** are a mechanism for one-way asynchronous notifications.
- The Linux kernel implements about 30 signals (the exact number is architecture-dependent).
- With the exception of **SIGKILL** (which always terminates the process), and **SIGSTOP** (which always stops the process), processes may control what happens when they receive a signal.
- Handled signals cause the execution of a user-supplied **signal handler** function.

Most important signals

SIGHUP (1)	Process's controlling terminal was closed (most frequently, the user logged out).	Terminate
SIGINT (2)	User generated the interrupt character (Ctrl-C).	Terminate
SIGABRT (6)	The **abort()** function sends this signal to the process that invokes it. The process then terminates and generates a core file.	Terminate with core dump
SIGKILL (9)	This signal is sent from the **kill()** system call; it exists to provide system administrators with a surefire way of unconditionally killing a process.	Terminate
SIGSEGV (11)	This signal, whose name derives from *segmentation violation*, is sent to a process when it attempts an invalid memory access.	Terminate with core dump
SIGTERM (15)	This signal is sent only by **kill()**; it allows a user to gracefully terminate a process (the default action).	Terminate

Most important signals (2)

SIGCHLD (17)	Whenever a process terminates or stops, the kernel sends this signal to the process' parent. A handler for this signal generally calls **wait()** to determine the child's pid and exit code.	Ignored
SIGCONT (18)	The kernel sends this signal to a process when the process is resumed after being stopped (by SIGSTOP).	Ignored
SIGSTOP (19)	This signal is sent only by **kill()**. It unconditionally stops a process, and cannot be caught or ignored.	Stop
SIGTSTP (20)	The kernel sends this signal to all processes in the foreground process group when the user provides the suspend character (usually **Ctrl-Z**).	Stop
SIGIO (29)	This signal is sent when an asynchronous I/O event is generated.	Terminate
SIGUSR1 (10) and SIGUSR2 (12)	These signals are available for user-defined purposes; the kernel never raises them. Processes may use **SIGUSR1** and **SIGUSR2** for whatever purpose they like.	Terminate

Signal management in C

- **sigemptyset**() - initializes an empty signal set, with all signals excluded from the set.
- **sigfillset**() - initializes a full signal set, including all signals.
- **sigaddset**() - adds a signal to a set.
- **sigdelset**() - removes a signal from a set.
- **sigprocmask**() - fetches and/or changes the signal mask of the calling thread.
- **sigaction**() - changes a signal action.
- **sigwait**() - wait for a signal.
- **strsignal**() - return string describing signal.

Managing Signals

```c
#include <signal.h>

int sigemptyset (sigset_t *set);
int sigaddset (sigset_t *set, int signo);
int sigprocmask (int how, const sigset_t *set, sigset_t
*oldset);

int sigaction (int signo, const struct sigaction *act,
struct sigaction *oldact);

struct sigaction {
    void     (*sa_handler)(int);
    void     (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t   sa_mask;
    int        sa_flags;
    void     (*sa_restorer)(void);
};
```

The behavior of **sigprocmask**( ) depends on the value of **how**, which is one of the following:

- **SIG_SETMASK** - the signal mask for the invoking process is changed to **set**.
- **SIG_BLOCK** - The signals in **set** are added to the invoking process' signal mask.
- **SIG_UNBLOCK** - The signals in **set** are removed from the invoking process' signal mask.

struct sigaction:

- **sa_handler** - address of the handler, **SIG_IGN** or **SIG_DFL**;
- **sa_mask** - signals to block;
- **sa_flags** - additional flags, like **SA_RESETHAND**
- SA_RESETHAND - enables "one-shot" mode. The behavior of the given signal is reset to the default once the signal handler returns.

Handle a signal

- Define the **SIGTERM** handler function
- Init a signal set, only **SIGHUP** is on and block the signal processing
- Set the handler for **SIGTERM**
- Print the **pid** of the process
- Enter endless loop

```c
void term_handler(int i) {
  printf ("Terminating\n");
  exit(EXIT_SUCCESS);
}

int main(int argc, char ** argv) {
  struct sigaction sa;
  sigset_t newset;

  sigemptyset(&newset);
  sigaddset(&newset, SIGHUP);
  sigprocmask(SIG_BLOCK, &newset, 0);

  sa.sa_handler = term_handler;
  sigaction(SIGTERM, &sa, 0);

  printf("My pid is %i\n", getpid());
  printf("Waiting...\n");

  while(1) sleep(1);

  return EXIT_FAILURE;
}
```

Waiting for Signals

```
#include <signal.h>

int sigwait(const sigset_t *set, int *sig);
```

- The **sigwait()** function suspends execution of the calling thread until one of the signals specified in the signal set **set** becomes pending.
- The function accepts the signal (removes it from the pending list of signals), and returns the signal number in **sig**.
- On success, **sigwait**() returns **0**. On error, it returns a positive error number.

Wait for signal

- Define the **SIGTERM** handler function
- Init a signal set, only **SIGHUP** is on and block the signal processing
- Set the handler for **SIGTERM**
- Print the **pid** of the process
- Wait for incoming **SIGHUP**s and notify
- If error occured exit with failure

```
...
  printf("My pid is %i\n", getpid());
  printf("Waiting...\n");

  while(!sigwait(&newset, &sig))
    printf("SIGHUP recieved\n");

  return EXIT_FAILURE;
}
```

Examine Signals

```
#include <string.h>
#include <signal.h>

char *strsignal(int sig);
extern const char * const sys_siglist[];

int sigfillset(sigset_t *set);
int sigdelset(sigset_t *set, int signum);
```

- **sys_siglist** is an array of strings holding the names of the signals supported by the system, indexed by signal number.
- A call to **strsignal( )** returns a pointer to a description of the signal given by **signo**.

Show signals info

- Define the **SIGTERM** handler function
- Init a full signal set, with only **SIGTERM** excluded
- Set the handler for **SIGTERM**
- Print the **pid** of the process
- Wait and show info for the incoming signals
- If error occured exit with failure

siginfo.c

```
void term_handler(int sig) {
  printf("Signal %i - %s\n", sig, sys_siglist[sig]);
  exit(EXIT_SUCCESS);
}
...
int main(int argc, char ** argv) {
...
  sigfillset(&sset);
  sigdelset(&sset, SIGTERM);
  sigprocmask(SIG_SETMASK, &sset, 0);
...
  while(!sigwait(&sset, &sig)) {
    printf("Signal %i - %s\n", sig, sys_siglist[sig]);
    printf("%s\n", strsignal(sig));
  }
...
```

Reentrancy

When the kernel raises a signal, a process can be executing code anywhere. The process might even be handling another signal. Signal handlers cannot tell what code the process is executing when a signal hits- the handler can run in the middle of anything. It is thus very important that any signal handler your process installs is a **reentrant** function.

A **reentrant** function is a function that is safe to call from within itself (or concurrently, from another thread in the same process). In order to qualify as reentrant, a function must not manipulate static data, must manipulate only stack-allocated data or data provided to it by the caller, and must not invoke any non-reentrant function.

Reentrancy

The standard C reentrant functions, which are safe to use:

accept, access, aio_error, aio_return, aio_suspend, alarm, bind, cfgetispeed, cfgetospeed, cfsetispeed, cfsetospeed, chdir, chmod, chown, clock_gettime, close, connect, creat, dup, dup2, execle, execve, _Exit & _exit, fchmod, fchown, fcntl, fdatasync, fork, fpathconf, fstat, fsync, ftruncate, getegid, geteuid, getgid, getgroups, getpeername, getpgrp, getpid, getppid, getsockname, getsockopt, getuid, kill, link, listen, lseek, lstat, mkdir, mkfifo, open, pathconf, pause, pipe, poll,posix_trace_event, pselect, raise, read, readlink, recv, recvfrom, recvmsg, rename, rmdir, select, sem_post, send, sendmsg,sendto, setgid, setpgid, setsid, setsockopt, setuid, shutdown, sigaction, sigaddset, sigdelset, sigemptyset, sigfillset,sigismember, signal, sigpause, sigpending, sigprocmask, sigqueue, sigset, sigsuspend, sleep, socket, socketpair, stat, symlink, sysconf, tcdrain, tcflow, tcflush, tcgetattr, tcgetpgrp, tcsendbreak, tcsetattr, tcsetpgrp, time, timer_getoverrun,timer_gettime, timer_settime, times, umask, uname, unlink, utime, wait, waitpid, write.

Anonymous and Named Pipes

- In computer science, an **anonymous pipe** is a simplex FIFO communication channel that may be used for **one-way** interprocess communication (IPC).
- Typically a parent program opens **anonymous pipes**, and creates a new process that inherits the other ends of the pipes.
- An **anonymous pipe** lasts only as long as the process lives.
- A **named pipe** (FIFO) is an extension to the traditional (anonymous) pipe concept.
- A **named pipe** can be identified by a name and appear as a file in the system.
- A **named pipe** can last as long as the system is up, beyond the life of the process. It can be deleted if no longer used.

Working with pipes

```c
#include <unistd.h>

int pipe(int pipefd[2]);

#include <sys/types.h>
#include <sys/stat.h>

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

pipe() - creates anonymous pipe.

- The array **pipefd** is used to return two file descriptors referring to the ends of the pipe.
- **pipefd[0]** refers to the **read** end of the pipe, **pipefd[1]** refers to the **write** end.

mkfifo() - makes a named pipe (FIFO).

- **mode** specifies the FIFO's permissions. It is modified by the process's **umask** in the usual way: the permissions of the created file are (**mode & ~umask**).

Anonymous pipe usage

passmsg.c

Open a pipe and store the I/O **fd**s

Fork the process

The parent: close the **input/read fd**

Write a message to the **output fd** and closes it

The child: close the **output/write fd**

Read a message from the **input fd**

Print the message to **stdout** and close **input fd**

```c
int main (int argc, char * argv[]){
  int pipedes[2];
  pid_t pid;
  pipe(pipedes);
  pid = fork();
  if ( pid > 0 ){
    char  *str = "String passed via pipe\n";
    close(pipedes[0]);
    write(pipedes[1], (void *) str, strlen(str) + 1);
    close(pipedes[1]);
  } else{
    char buf[1024];
    int len;
    close(pipedes[1]);
    while ((len = read(pipedes[0], buf, 1024)) != 0)
      write(2, buf, len);
    close(pipedes[0]);
  }
  return 0;
}
```

Chat server

- Make a FIFO named '**./fifofile**'
- Open for write '**./fifofile**' and exit on error
- Do until we receive '**q**' from the keyboard
  - Read a symbol (**key**) and write it to the FIFO file
  - Flush the FIFO file if the **key** was Enter
- Close and delete the FIFO file

chatsrv.c

```c
#define FIFO_NAME "./fifofile"
int main(int argc, char * argv[]) {
  FILE * f;
  char ch;
  mkfifo(FIFO_NAME, 0600);
  f = fopen(FIFO_NAME, "w");
  if (f == NULL){
    printf("Faile to open FIFO!\n");
    return -1;
  }

  do {
    ch = getchar();
    fputc(ch, f);
    if (ch == 10) fflush(f);
  } while (ch != 'q');

  fclose(f);
  unlink(FIFO_NAME);
  return 0;
}
```

chatclnt.c

Chat client

Open for read the file named '**./fifofile**'

Do until we receive '**q**' from the keyboard

Read a symbol (**key**)

Write the received symbol to the console

Close and delete the FIFO file

```c
#define FIFO_NAME "./fifofile"
int main ()
{
  FILE * f;
  char ch;
  f = fopen(FIFO_NAME, "r");
  do
  {
    ch = fgetc(f);
    putchar(ch);
  } while (ch != 'q');
  fclose(f);
  unlink(FIFO_NAME);
  return 0;
}
```

Exercise

Program *ReverseEncryptorDecryptor*:

Let's define "Reverse Encryption" (**RE**) as a method to secure text messages - the messages are simply reverted, before they are sent further. Write a program ('**revendec.c**'), which have 2 processes communicating through an **anonymous pipe**:

- Process 1 should read text lines from the keyboard, then apply **RE** to it and send the encrypted message to Process 2.
- Process 2 should await for incoming messages, print the encrypted messages to the screen, then decrypt them and store into a file ('**messages.log**').

Project *EncryptedChat*:

Write a server/client pair of programs ('**echatsrv.c**' and '**echatclnt.c**'). The server should mimic the behaviour of Process 1 from the previous task and the client - Process 2, respectively. The programs should use a **FIFO** (named pipe) as a communication method between them.