# Взаимодействие между процессами

## Process ID

```c
#include <sys/types.h>
#include <unistd.h>

typedef int pid_t;

pid_t getpid();
pid_t getppid();
uid_t getuid();
gid_t getgid();
```

---

```
$ cat /proc/sys/kernel/pid_max
4194304
```

## fork

fork(2)

```c
#include <unistd.h>

pid_t fork();
```

Дочерний процесс получает копии:

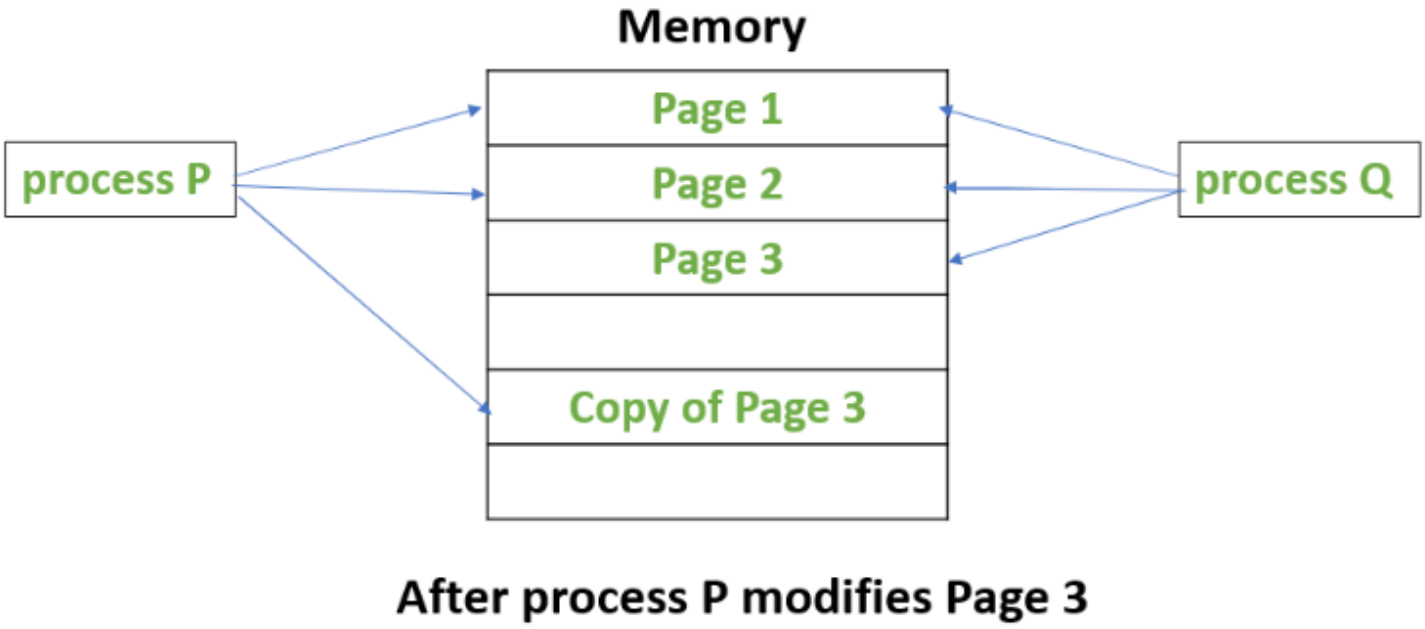
- сегмента данных,
- кучи,
- стека.

Сегмент кода используется совместно.

Dennis M. Ritchie, The Evolution of the Unix Time-sharing System

```c
#include <stdio.h>
#include <unistd.h>

int globvar = 6;

int main()
{
    int var;
    pid_t pid;

    var = 88;
    printf("перед вызовом функции fork\n");

    if ((pid = fork()) < 0) {
        perror("fork");
    } else if (pid == 0) {
        globvar++;
        var++;
    } else {
        sleep(2);
    }
    printf("pid = %ld, globvar = %d, var = %d\n",
        (long)getpid(), globvar, var);
    return 0;
}
```

Copy-on-write




Dirty CoW: CVE-2016-5195

Также наследуются:

- файловые дескрипторы (включая позицию),
- uid и gid,
- идентификаторы группы процессов и сеанса,
- управляющий терминал,
- cwd и root,
- umask,
- маска сигналов,
- переменные окружения,
- shared memory,
- отображения в память,
- ограничения на ресурсы.

Различаются:

- pid и ppid,
- значения times(2),
- блокировки файлов (fcntl(2)) сбрасываются,
- таймеры сбрасываются,
- сигналы, ожидающие обработки, очищаются.

### exec

exec(3)

```
#include <unistd.h>
int execl(const char* pathname, const char* arg0, ... /* (char*)0 */);
int execv(const char* pathname, char* const argv[]);
int execle(const char* pathname, const char* arg0, ...
    /* (char *)0, char* const envp[] */ );
int execve(const char* pathname, char* const argv[], char* const envp[]);
int execlp(const char* filename, const char* arg0, ... /* (char*)0 */ );
int execvp(const char* filename, char* const argv[]);
int fexecve(int fd, char* const argv[], char* const envp[]);
```

close-on-exec flag:

- O_CLOEXEC in open(2)

- FD_CLOEXEC in fcntl(2)

### vfork

vfork(2)

> The vfork() function has the same effect as fork(2), except that the behavior is undefined if the process created by vfork() either modifies any data other than a variable of type pid_t used to store the return value from vfork(), or returns from the function in which vfork() was called, or calls any other function before successfully calling _exit(2) or one of the exec(3) family of functions.
>
> CONFORMING TO: 4.3BSD; POSIX.1-2001 (but marked OBSOLETE). POSIX.1-2008 removes the specification of vfork().

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int globvar = 6;

int main()
{
    int var;
    pid_t pid;

    var = 88;
    printf("перед вызовом функции vfork\n");
    if ((pid = vfork()) < 0) {
        perror("fork");
    } else if (pid == 0) {
        globvar++;
        var++;
        exit(0);
    }
    printf("pid = %ld, globvar = %d, var = %d\n",
        (long)getpid(), globvar, var);
    return 0;
}
```

clone(2), rfork(2)

unshare(2)

posix_spawn(3)

fork() is evil; vfork() is goodness; afork() would be better; clone() is stupid, перевод

### system

system(3)

```c
#include <stdlib.h>

int system(const char* cmdstring);
```

### wait

wait(2)

```c
#include <sys/wait.h>

pid_t wait(int* statloc);
pid_t waitpid(pid_t pid, int* statloc, int options);
int waitid(idtype_t idtype, id_t id, siginfo_t* infop, int options);

/* options */
```

```
WNOHANG;
WUNTRACED;
WCONTINUED;

/* status */
WIFEXITED;
WEXITSTATUS;
WIFSIGNALED;
WTERMSIG;
WCOREDUMP;
WIFSTOPPED;
WSTOPSIG;
WIFCONTINUED;

/* idtype */
P_PID;
P_PGID;
P_ALL;
```

- Если родительский процесс завершается раньше дочернего, родителем последнего становится `init`.
- Если дочерний процесс завершается до того, как родитель вызывает `wait` или `waitpid` для его pid, такой процесс становится “зомби”.

```c
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main ()
{
  pid_t child_pid;

  child_pid = fork ();
  if (child_pid > 0) {
    sleep (60);
  }
  else {
    exit (0);
  }
  return 0;
}
```

Изучаем процессы в Linux

### Race condition

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

static void charatatime(char *str)
{
    char *ptr;
    int c;

    setbuf(stdout, NULL);
    for (ptr = str; (c = *ptr++) != 0; )
        putc(c, stdout);
}

int main()
{
    pid_t pid;
```

```
    if ((pid = fork()) < 0) {
        perror("fork");
    } else if (pid == 0) {
        charatatime("from child process\n");
    } else {
        charatatime("from parent process\n");
    }
    exit(0);
}
```

### Сигналы

signal(7)

- null signal: сигнал с номером 0.
- Сигналы, генерируемые терминалом (SIGINT, SIGHUP).
- Аппаратные ошибки (SIGSEGV, SIGBUS).
- Программные ситуации (SIGURG, SIGPIPE, SIGALRM).

```
/* ISO C99 signals.  */
#define SIGINT          2       /* Interactive attention signal.  */
#define SIGILL          4       /* Illegal instruction.  */
#define SIGABRT         6       /* Abnormal termination.  */
#define SIGFPE          8       /* Erroneous arithmetic operation.  */
#define SIGSEGV         11      /* Invalid access to storage.  */
#define SIGTERM         15      /* Termination request.  */

/* Historical signals specified by POSIX. */
#define SIGHUP          1       /* Hangup.  */
#define SIGQUIT         3       /* Quit.  */
#define SIGTRAP         5       /* Trace/breakpoint trap.  */
#define SIGKILL         9       /* Killed.  */
#define SIGPIPE         13      /* Broken pipe.  */
#define SIGALRM         14      /* Alarm clock.  */

/* Historical signals specified by POSIX. */
#define SIGBUS           7      /* Bus error.  */
#define SIGSYS          31      /* Bad system call.  */

/* New(er) POSIX signals (1003.1-2008, 1003.1-2013).  */
#define SIGURG          23      /* Urgent data is available at a socket.  */
#define SIGSTOP         19      /* Stop, unblockable.  */
#define SIGTSTP         20      /* Keyboard stop.  */
#define SIGCONT         18      /* Continue.  */
#define SIGCHLD         17      /* Child terminated or stopped.  */
#define SIGTTIN         21      /* Background read from control terminal.  */
#define SIGTTOU         22      /* Background write to control terminal.  */
#define SIGPOLL         29      /* Pollable event occurred (System V).  */
#define SIGXFSZ         25      /* File size limit exceeded.  */
#define SIGXCPU         24      /* CPU time limit exceeded.  */
#define SIGVTALRM       26      /* Virtual timer expired.  */
#define SIGPROF         27      /* Profiling timer expired.  */
#define SIGUSR1         10      /* User-defined signal 1.  */
#define SIGUSR2         12      /* User-defined signal 2.  */
```

---

```
#include <signal.h>

void psignal(int sig, const char* msg);
void psiginfo(const siginfo_t *pinfo, const char* msg);
```

kill(2), raise(3)

```c
#include <signal.h>

int kill(pid_t pid, int sig);

int raise(int signo);
```

Default disposition

- Term
- Ign
- Core
- Stop
- Cont

signal(2)

```c
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);

/* handler */
SIG_IGN;
SIG_DFL;
```

---

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

static void sig_usr(int signo)
{
    if (signo == SIGUSR1)
        printf("принят сигнал SIGUSR1\n");
    else if (signo == SIGUSR2)
        printf("принят сигнал SIGUSR2\n");
    else
        printf("принят сигнал %d\n", signo);
}

int main()
{
    if (signal(SIGUSR1, sig_usr) == SIG_ERR)
        perror("signal SIGUSR1");
    if (signal(SIGUSR2, sig_usr) == SIG_ERR)
        perror("signal SIGUSR2");
    for(;;)
        pause();
}
```

pause(2)

```c
#include <unistd.h>

int pause();
```

Правила использования сигналов в Unix

О чем нельзя забывать при работе с POSIX-сигналами

**Прерванные сигналами системные вызовы**

signal(7)

> If a signal handler is invoked while a system call or library function call is blocked, then either: * the call is automatically restarted after the signal handler returns; or * the call fails with the error EINTR.

- `read(2)`, `write(2)`, `ioctl(2)` on “slow” device: terminal, pipe, socket.
- `open(2)` if it can block (e.g. for a FIFO).
- `wait(2)`, `waitpid(2)`.
- socket interfaces: `accept(2)`, `connect(2)`, `send(2)`, `recv(2)`.
- `flock(2)` and `fcntl(2)`.
- POSIX message queue interfaces.
- `getrandom(2)`.
- `pthread_mutex_lock(3)`, `pthread_cond_wait(3)` and related.
- POSIX semaphore interfaces.
- `read(2)` from `inotify(7)`.

```
again:
    if ((n = read(fd, buf, BUFFSIZE)) < 0) {
        if (errno == EINTR)
            goto again; /* просто прерванный системный вызов */
                        /* обработать другие возможные ошибки */
}
```

The Rise of Worse is Better by Richard P. Gabriel, перевод

> Unix and C are the ultimate computer viruses.

#### Реентрабельные функции

signal-safety(7)

> Fetching and setting the value of errno is async-signal-safe provided that the signal handler saves errno on entry and restores its value before returning.

alarm(2)

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

getpwnam(3)

```
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <pwd.h>
#include <unistd.h>

static void my_alarm(int signo)
{
    struct passwd *rootptr;
    printf("внутри обработчика сигнала\n");
    if ((rootptr = getpwnam("root")) == NULL)
        perror("getpwnam");
    alarm(1);
}

static const char* username = "andrew";

int main(void)
{
    struct passwd* ptr;
    signal(SIGALRM, my_alarm);
    alarm(1);
    for (;;) {
        if ((ptr = getpwnam(username)) == NULL)
            perror("getpwnam");
        if (strcmp(ptr->pw_name, username) != 0)
```

```
            printf("возвращаемое значение повреждено! pw_name = %s\n",
                    ptr->pw_name);
    }
}
```

#### Маска заблокированных сигналов

```
#include <signal.h>

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);

int sigismember(const sigset_t *set, int signo);
```

sigprocmask(2)

```
#include <signal.h>

int sigprocmask(int how, const sigset_t* set, sigset_t* oset);

/* how */
SIG_BLOCK;
SIG_UNBLOCK;
SIG_SETMASK;
```

#### sigaction

sigaction(2)

```
#include <signal.h>

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

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

### Межпроцессовые коммуникации

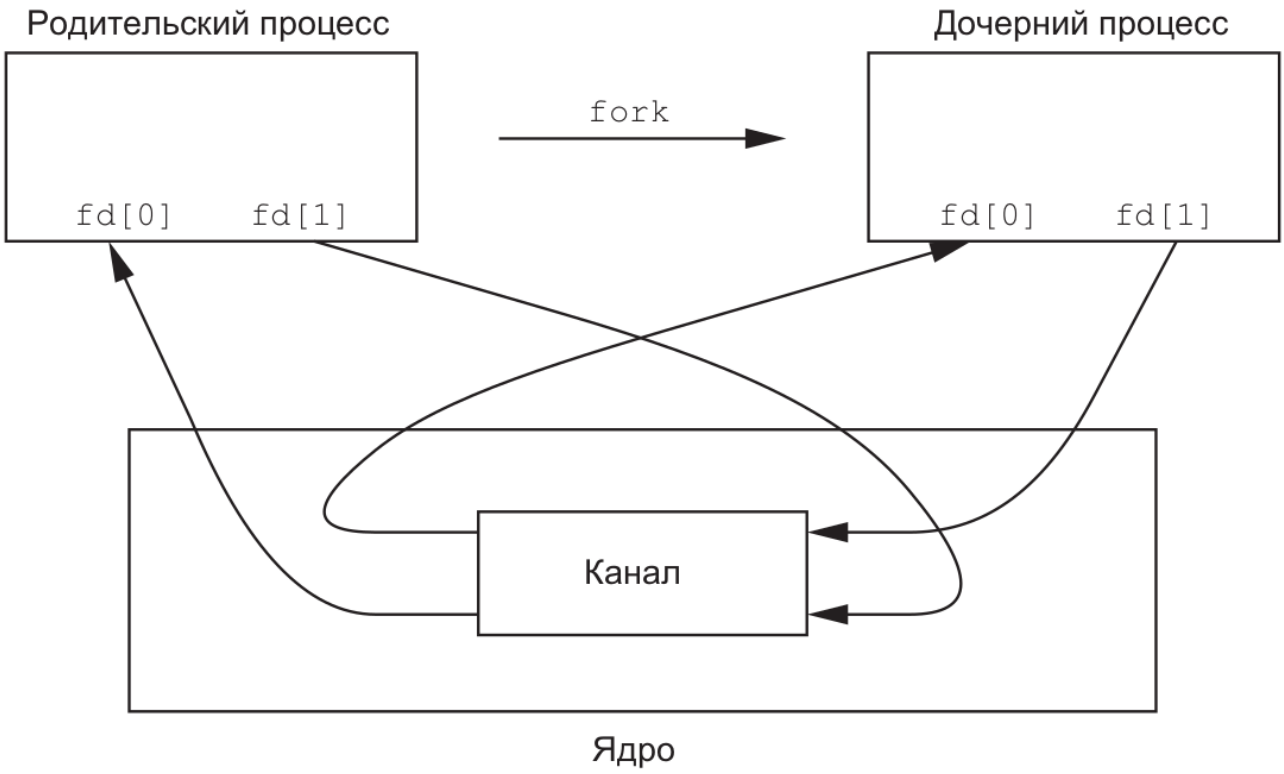
Межпроцессовые коммуникации LINUX

#### Полудуплексные каналы (pipe)

```
ls | sort
```

pipe(2)

```
#include <unistd.h>

int
pipe(int pipefd[2]);

int
pipe2(int pipefd[2], int flags);

#define PIPE_BUF 4096  /* Linux */
```




```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(void)
{
  int     fd[2], nbytes;
  pid_t   childpid;
  char    string[] = "Hello, world!\n";
  char    readbuffer[80];

  pipe(fd);
  if ((childpid = fork()) == -1)
  {
     perror("fork");
     exit(1);
  }
  if (childpid == 0)
  {
     /* Child process closes up input side of pipe */
     close(fd[0]);
     /* Send "string" through the output side of pipe */
     write(fd[1], string, strlen(string));
     exit(0);
  }
  else
  {
    /* Parent process closes up output side of pipe */
    close(fd[1]);
    /* Read in a string from the pipe */
    nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
    printf("Received string: %s", readbuffer);
  }
  return(0);
}
```

pipe(7)

> A pipe has a limited capacity. If the pipe is full, then a write(2) will block or fail, depending on whether the O_NONBLOCK flag is set. Different implementations have different limits for the pipe capacity. Applications should not rely on a particular capacity: an application should be designed so that a reading process consumes data as soon as it is available, so that a writing process does not remain blocked.
>
> Since Linux 2.6.11, the pipe capacity is 16 pages (i.e., 65,536 bytes in a system with a page size of 4096 bytes).

Так все же, насколько быстры каналы в Linux?

Benchmarks for Inter-Process-Communication Techniques

**popen**

popen(3)

```
#include <stdio.h>

FILE *popen(const char* cmdstring, const char* type);

int pclose(FILE *stream);
```

**Именованные каналы (FIFO)**

```
$ mknod MYFIFO p
```

mknod(2)

```
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int mknod(const char *pathname, mode_t mode, dev_t dev);
```

---

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>
#include <linux/stat.h>

#define FIFO_FILE       "MYFIFO"

int main(void)
{
  FILE *fp;
  char readbuf[80];

  /* Create the FIFO if it does not exist */
  umask(0);
  mknod(FIFO_FILE, S_IFIFO|0666, 0);
  while(1)
  {
    fp = fopen(FIFO_FILE, "r");
    fgets(readbuf, 80, fp);
    printf("Received string: %s\n", readbuf);
    fclose(fp);
  }
  return(0);
}
```

**UNIX domain sockets**

unix(7)

unix domain sockets vs. internet sockets

```c
#include <sys/socket.h>

int socketpair(int domain, int type, int protocol, int sockfd[2]);
```

---

```c
/* Linux, Solaris */
struct sockaddr_un {
    sa_family_t sun_family; /* AF_UNIX */
    char sun_path[108];     /* полное имя */
};

/* FreeBSD, MacOS */
struct sockaddr_un {
    unsigned char sun_len;  /* длина адреса сокета */
    sa_family_t sun_family; /* AF_UNIX */
    char sun_path[104];     /* полное имя */
};
```

---

```c
#include <stdio.h>
#include <stddef.h>
#include <sys/socket.h>
#include <sys/un.h>

int main()
{
    int fd, size;
    struct sockaddr_un un;

    un.sun_family = AF_UNIX;
    strcpy(un.sun_path, "foo.socket");
    if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
        perror("socket");
    size = offsetof(struct sockaddr_un, sun_path) + strlen(un.sun_path);
    if (bind(fd, (struct sockaddr *)&un, size) < 0)
        perror("bind");

    printf("Done!\n");
    return 0;
}
```

**UDS server**

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>


#define NAME "echo.socket"

int main()
{
    int sock, msgsock, rval;
    struct sockaddr_un server;
    char buf[1024];

    sock = socket(AF_UNIX, SOCK_STREAM, 0);
```

```
    if (sock < 0) {
        perror("opening stream socket");
        exit(1);
    }
    server.sun_family = AF_UNIX;
    strcpy(server.sun_path, NAME);
    if (bind(sock, (struct sockaddr *) &server, sizeof(struct sockaddr_un))) {
        perror("binding stream socket");
        exit(1);
    }
    printf("Socket has name %s\n", server.sun_path);
    listen(sock, 5);
    for (;;) {
        msgsock = accept(sock, 0, 0);
        if (msgsock == -1)
        {
            perror("accept");
            break;
        }
        else do {
            bzero(buf, sizeof(buf));
            if ((rval = read(msgsock, buf, 1024)) < 0)
                perror("reading stream message");
            else if (rval == 0)
                printf("Ending connection\n");
            else
                send(msgsock, buf, strlen(buf), 0);
        } while (rval > 0);
        close(msgsock);
    }
    close(sock);
    unlink(NAME);
}
```

Windows/WSL Interop with AF_UNIX