Сигналы, процессы, сокеты в языке Си

Подготовительное отделение C/C++ (открытый курс)

образование

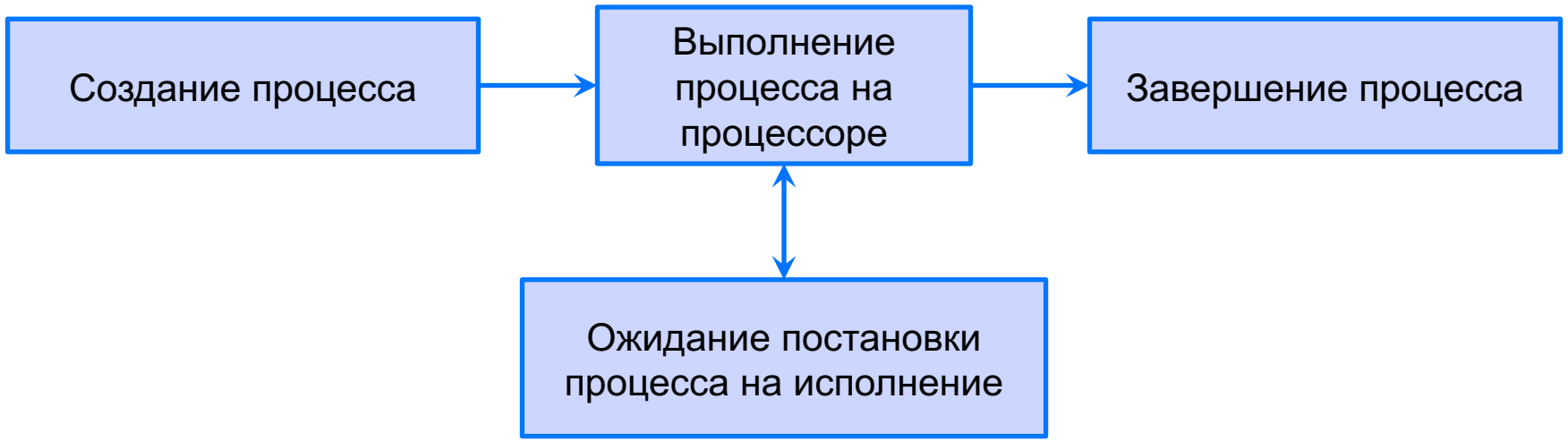
Функции операционной системы

- **Управление процессами**: формирование процессов, поддержание жизненного цикла процесса, организация взаимодействия процессов и работы процессов с ресурсами.
- **Управление оперативной памятью**: поддержка аппарата виртуальной памяти, защита оперативной памяти от несанкционированного доступа, проверка корректности работы процесса с выделенной ему оперативной памятью.
- **Планирование**: планирование доступа процессов к центральному процессору, организация и обработка очередей обмена, обработка прерываний.
- **Управление данными, файловой системой и устройствами**: поддержание структуры файловой системы, обеспечение обмена данными и управление устройствами, такими как диск, сетевая карта и т. д.
- **Сетевое взаимодействие**: реализация и обеспечение функционирования сетевых протоколов.

Управление процессами

Процесс — совокупность **машинных команд** и **данных**, обрабатываемая в вычислительной системе и обладающая правами на владение некоторым набором **ресурсов** вычислительной системы.

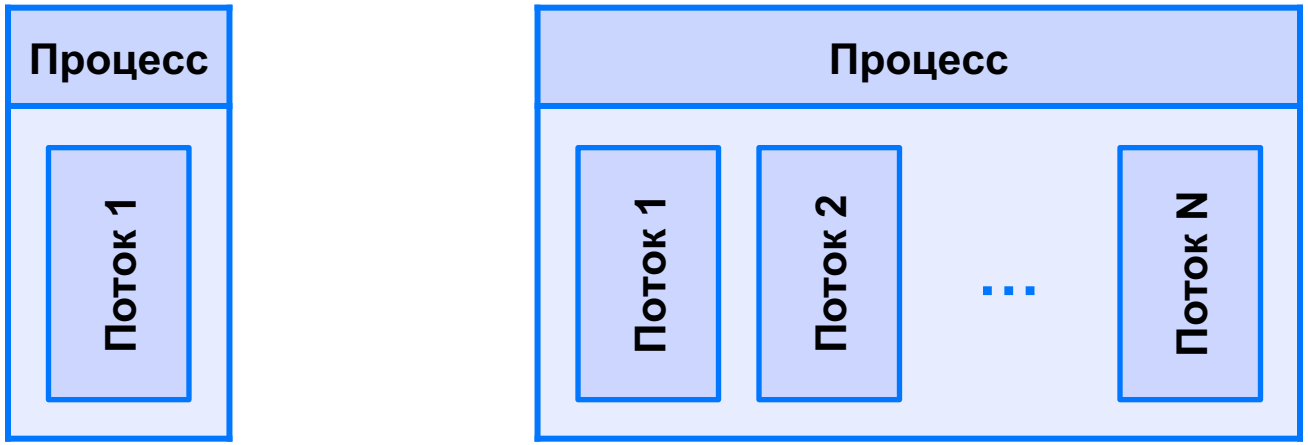
Жизненный цикл процесса:



Типы процессов

Процесс (или **полновесный процесс**) – является объектом планирования и выполняется внутри защищённой области памяти.

Потоки (или **легковесные процессы**, известные также как **нити**) — это процессы, которые могут активироваться внутри полновесного процесса, могут быть объектами планирования, и при этом они могут функционировать внутри общей (т.е. незащищённой от других потоков) области памяти.



Контекст процесса

- пользовательская составляющая:
  - сегмент кода:
    - машинные команды;
    - неизменяемые константы;
  - сегмент данных:
    - область статической памяти;
    - thread-local storage;
    - область разделяемой памяти;
    - область стека;
    - динамическая память;
- аппаратная составляющая:
  - актуальное состояние регистров;
  - таблица страниц виртуальной памяти;
- системная составляющая:
  - информация идентификационного характера:
    - PID процесса;
    - PID родительского процесса;
  - сохранённая копия аппаратной составляющей;
  - информация, необходимая для управления процессом:
    - состояние процесса;
    - таблица дескрипторов открытых файлов процесса;
    - обработчики сигналов;
    - информация сигналах, ожидающих доставки в процесс;
    - реальный и эффективный идентификаторы пользователей-владельцев;
    - реальный идентификатор групп пользователей-владельцев;
    - окружение;
    - приоритет.

Окружение процесса в Unix

```cpp
#include <iostream>


int main(int argc, char** argv, char** envp) {
    std::cout << "ARGV:" << std::endl;
    for (int i = 0; argv[i] != NULL; ++i)
        std::cout << "\t" << argv[i] << std::endl;


    std::cout << "ENV:" << std::endl;
    for (int i = 0; envp[i] != NULL; ++i)
        std::cout << "\t" << envp[i] << std::endl;
    return 0;
}
```

```
[i.anferov@hostname ~]$ ./a.out arg1 arg2 arg3
ARGV:
    ./a.out
    arg1
    arg2
    arg3
ENV:
    USER=i.anferov
    HOME=/Users/i.anferov
    PWD=/Users/i.anferov
    PATH=/usr/bin:/bin:/usr/sbin:/sbin
    SHELL=/bin/bash
[i.anferov@hostname ~]$
```

```cpp
char* getenv  (const char *name);
int   setenv  (const char *name, const char *value, int overwrite);
int   unsetenv(const char *name);
```

Процессы в операционной системе Unix. Системный вызов fork()

Процесс в операционной системе Unix порождается при помощи системного вызова **fork()** (исключения — процессы с PID 0 и 1).

```cpp
#include <unistd.h>
#include <iostream>

int main(int argc, char** argv, char** envp) {
    std::cout << getpid() << ": Before fork" << std::endl;
    fork();
    std::cout << getpid() << ": After fork" << std::endl;

    return 0;
}
```

```
[i.anferov@hostname ~]$ ./a.out
99759: Before fork
99759: After fork
99762: After fork
[i.anferov@hostname ~]$
```

```cpp
#include <unistd.h>
#include <iostream>

int main(int argc, char** argv, char** envp) {
    std::cout << getpid() << ": Before fork\n";
    fork();
    std::cout << getpid() << ": After fork\n";

    return 0;
}
```

```
[i.anferov@hostname ~]$ ./a.out | cat
321: Before fork
321: After fork
321: Before fork
323: After fork
[i.anferov@hostname ~]$
```

Процессы в операционной системе Unix. Системный вызов fork()

```
#include <unistd.h>
#include <iostream>

int main(int argc, char** argv, char** envp) {
    std::cout << getpid() << ": Before fork" << std::endl;

    if (int ret = fork())
        std::cout << getpid() << ": After fork in parent. Child PID is " << ret << std::endl;
    else
        std::cout << getpid() << ": After fork in child. Parent PID is " << getppid() << std::endl;

    return 0;
}
```

```
[i.anferov@hostname ~]$ ./a.out
583: Before fork
583: After fork in parent. Child PID is 606
606: After fork in child. Parent PID is 583
[i.anferov@hostname ~]$
```

Замена тела процесса. Семейство функций exec*

```c
#include <unistd.h>

int execl(const char *path, char *arg0,.../*, NULL*/);
int execlp(const char *file, char *arg0,.../*, NULL*/);
int execle(const char *path, char *arg0, .../*, NULL*/, char *const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(const char *path, char *const argv[], char *const envp[]);
```

```
[i.anferov@hostname ~] $ which echo
/bin/echo
[i.anferov@hostname ~]$
```

```
[i.anferov@hostname ~]$ echo $PATH
/usr/bin:/bin:/usr/sbin:/sbin
[i.anferov@hostname ~]$
```

Замена тела процесса

```
[i.anferov@hostname ~]$ ./a.out
3159 ARGV:
3159    print_args_and_env
3159    arg1
3159    arg2
3159 ENV:
3159    SHELL=/bin/bash
3160 ARGV:
3160    print_args_and_env
3160    arg1
3160    arg2
3160 ENV:
3159    USER=i.anferov
3160    VAR1=value1
3160    VAR2=value2
3159    PATH=/usr/bin:/bin:/usr/sbin:/sbin
3159    PWD=/Users/i.anferov
3159    HOME=/Users/i.anferov
[i.anferov@hostname ~]$
```

```cpp
int main(int argc, char** argv, char** envp) {
    std::cout << getpid() << " ARGV:" << std::endl;
    for (int i = 0; argv[i] != NULL; ++i)
        std::cout << getpid() << "\t" << argv[i] << std::endl;

    std::cout << getpid() << " ENV:" << std::endl;
    for (int i = 0; envp[i] != NULL; ++i)
        std::cout << getpid() << "\t" << envp[i] << std::endl;

    return 0;
}
```

```cpp
int main(int argc, char** argv, char** envp) {
    if (fork()) {
        execl("print_args_and_env", "print_args_and_env", "arg1", "arg2", NULL);
        perror("exec in parent failed");
    } else {
        const char* args[] = { "print_args_and_env", "arg1", "arg2", NULL };
        const char* env[] = { "VAR1=value1", "VAR2=value2", NULL };
        execve("print_args_and_env", (char**)args, (char**)env);
        perror("exec in child failed");
    }

    return -1;
}
```

Файловые дескрипторы

```c
#include <fcntl.h>

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

```
O_RDONLY  open for reading only
O_WRONLY  open for writing only
O_RDWR    open for reading and writing
```

```
O_APPEND  append on each write
O_CREAT   create file if it does not exist
O_TRUNC   truncate size to 0
O_EXCL    error if O_CREAT and the file exists
```

```
S_IRUSR  00400 user has read permission
S_IWUSR  00200 user has write permission
S_IXUSR  00100 user has execute permission
S_IRGRP  00040 group has read permission
S_IWGRP  00020 group has write permission
S_IXGRP  00010 group has execute permission
S_IROTH  00004 others have read permission
S_IWOTH  00002 others have write permission
S_IXOTH  00001 others have execute permission
```

```c
#include <unistd.h>

ssize_t read (int fd,       void *buf, size_t nbyte);
ssize_t write(int fd, const void *buf, size_t nbyte);
int     close(int fd);
int     dup2 (int from_fd, int to_fd);
```

Предопределённые файловые дескрипторы

```c
int main() {
    char buf[1024];

    while (true) {
        ssize_t r = read(0, buf, sizeof(buf));

        if (r < 0) {
            perror("read from stdin failed"); return -1;
        }

        if (r == 0) // EOF reached
            return 0;

        if (write(1, buf, r) != r) {
            perror("write to stdout failed"); return -1;
        }

        for (int i = 0; i < r; ++i)
            buf[i] = toupper(buf[i]);

        if (write(2, buf, r) != r) {
            perror("write to stderr failed"); return -1;
        }
    }
}
```

```
$ cat hello_world.txt
Hello, World!
$ ./a.out 0<hello_world.txt 1>stdout.txt 2>stderr.txt
$ cat stdout.txt
Hello, World!
$ cat stderr.txt
HELLO, WORLD!
```

```
STDIN_FILENO   0
STDOUT_FILENO  1
STDERR_FILENO  2
```

Подмена файловых дескрипторов

```c
int main(int argc, char** argv) {
    assert(argc == 4);

    int stdin_fd = open(argv[2], O_RDONLY);
    if (stdin_fd == -1) {
        perror("failed to open input file"); return -1;
    }
    dup2(stdin_fd, 0);
    close(stdin_fd);

    int stdout_fd = open(argv[3], O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (stdout_fd == -1) {
        perror("failed to open output file"); return -1;
    }
    dup2(stdout_fd, 1);
    close(stdout_fd);

    execlp(argv[1], argv[1], NULL);

    perror("exec failed");
    return -1;
}
```

```
$ cat hello_world.txt
Hello, World!
$ ./a.out cat hello_world.txt stdout.txt
$ cat stdout.txt
Hello, World!
```

Взаимодействие процессов в ОС Unix. Сигналы

Сигнал — средство уведомления процесса о наступлении определённого события.

Сигналы — механизм **асинхронного** взаимодействия. Момент прихода сигнала заранее не известен.

Возможная реакция процесса на сигнал:

- обработка сигнала по умолчанию;
- перехват и обработка сигнала в заранее установленном процессом обработчике;
- игнорирование сигнала.

Сигналы `SIGKILL` и `SIGSTOP` не могут быть проигнорированы или обработаны процессом.

```c
#include <signal.h>

int raise(int sig);

// pid >   0:  sig is sent to the process whose ID is equal to pid
// pid ==  0:  sig is sent to all processes of the current group
// pid == -1:  sig is sent to all other processes
// pid <  -1:  sig is sent to all processes of the group with ID equal to pid
int kill(pid_t pid, int sig);
```

Обработка сигналов

```c
#include <signal.h>

struct sigaction {
    union {
        void (*sa_handler)(int signo);  // May be SIG_DFL or SIG_IGN
        void (*sa_sigaction)(int signum, siginfo_t*, void* ucontext);
    };
    sigset_t     sa_mask;
    int          sa_flags;
};

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);

SA_NODEFER   // Не маскировать сигнал перед запуском обработчика
SA_RESETHAND // Вызвать обработчик только для первого пришедшего сигнала
SA_RESTART   // Перезапустить прерванный системный вызов после завершения обработки сигнала
SA_SIGINFO   // Использовать обработчик sa_sigaction

int sigaction(int sig, const struct sigaction* new_action, struct sigaction* old_action);
int sigprocmask(int how /* SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK */, const sigset_t* mask, sigset_t* old_mask);
```

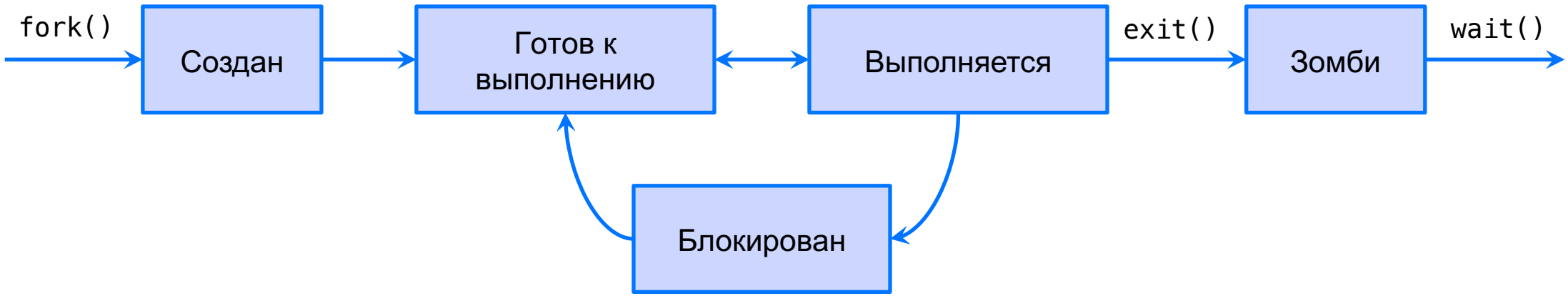
Сигналы

```
const char* msg;
int party_pid;
int done = 0;

void handle(int) {
    printf("%s\n", msg);
    sleep(1);
    kill(party_pid, SIGUSR1);
    if (++done >= 5)
        exit(0);
}
```

```
[i.anferov@hostname ~] $ ./a.out
ping
pong
ping
pong
ping
pong
ping
pong
ping
pong
```

```
int main() {
    struct sigaction action;
    action.sa_handler = handle;
    sigemptyset(&action.sa_mask);

    sigaction(SIGUSR1, &action, NULL);

    sigset_t sigusr1_set;
    sigemptyset(&sigusr1_set);
    sigaddset(&sigusr1_set, SIGUSR1);

    sigprocmask(SIG_BLOCK, &sigusr1_set, NULL);

    if (int pid = fork(); pid > 0) {
        msg = "ping";
        party_pid = pid;
        sigprocmask(SIG_UNBLOCK, &sigusr1_set, NULL);
        raise(SIGUSR1);
    } else {
        msg = "pong";
        party_pid = getppid();
        sigprocmask(SIG_UNBLOCK, &sigusr1_set, NULL);
    }

    for (;;);
}
```

Жизненный цикл процесса в Unix



Получение информации о завершении процесса

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

pid_t wait   (           int *status);
pid_t wait3  (           int *status, int options, struct rusage *rusage);
pid_t wait4  (pid_t pid, int *status, int options, struct rusage *rusage);
pid_t waitpid(pid_t pid, int *status, int options);
```

```
WIFEXITED(status)       // True if the process terminated normally by a call to _exit() or exit().
    WEXITSTATUS(status) // Evaluates to the argument passed to _exit() or exit() by the child.

WIFSIGNALED(status)     // True if the process terminated due to receipt of a signal.
    WTERMSIG(status)    // Evaluates to the number of the signal that caused the termination.
```

```
WNOHANG // option is used to indicate that the call should not block if there are no zombie children
```

Получение информации о завершении процесса

```c
int main() {
    if (int pid = fork(); pid > 0) { // parent
        if (int pid = fork(); pid > 0) { // still in parent
            while (true) {
                int status;
                if (int ret = wait(&status); ret == -1) {
                    if (errno == EINTR)
                        continue;
                    if (errno == ECHILD)
                        return 0;
                    perror("failed to wait()");
                } else { // ret variable has finished child PID
                    if (WIFEXITED(status))
                        printf("process %d finished with exit code %d\n", ret, WEXITSTATUS(status));
                    else if (WIFSIGNALED(status))
                        printf("process %d terminated with signal %d\n", ret, WTERMSIG(status));
                }
            }
        } else {
            printf("running 'ls ...' in process %d\n", getpid());
            execlp("ls", "ls", "./some_unexisting_file", NULL);
            perror("failed to run 'ls'");
            return -1;
        }
    } else {
        printf("raising SIGINT in process %d\n", getpid());
        raise(SIGINT);
    }
}
```

```
[i.anferov@hostname ~]$ ./a.out
raising SIGINT in process 8490
running 'ls ...' in process 8491
process 8490 terminated with signal 2
ls: ./some_unexisting_file: No such file or directory
process 8491 finished with exit code 1
```

Взаимодействие процессов в ОС Unix. Неименованные каналы

```
[i.anferov@hostname ~]$ yes hello | cat -n | head -n 10
     1  hello
     2  hello
     3  hello
     4  hello
     5  hello
     6  hello
     7  hello
     8  hello
     9  hello
    10  hello
```

```c
#include <unistd.h>

int pipe(int fd[2]); // fd[0] – out, fd[1] – in
```

Неименованные каналы

```c
int main(int argc, char** argv) {
    assert(argc == 3);

    int pipe_fds[2]; // fd[0] – out, fd[1] – in
    if (pipe(pipe_fds)) {
        perror("failed to create pipe"); return -1;
    }

    if (int ret = fork(); ret > 0) { // parent
        close(pipe_fds[0]);
        dup2(pipe_fds[1], 1);
        close(pipe_fds[1]);
        execlp(argv[1], argv[1], NULL);
        perror("exec 1 failed");
    } else if (ret == 0) { // child
        close(pipe_fds[1]);
        dup2(pipe_fds[0], 0);
        close(pipe_fds[0]);
        execlp(argv[2], argv[2], NULL);
        perror("exec 2 failed");
    } else { // ret < 0. ERROR while fork()
        perror("fork failed");
    }

    return -1;
}
```

```
[i.anferov@hostname ~]$ ./a.out yes head
y
y
y
y
y
y
y
y
y
y
```

Взаимодействие процессов в ОС Unix. Именованные каналы

```
[i.anferov@hostname ~]$ mkfifo -m 0644 /tmp/fifo100500
[i.anferov@hostname ~]$ yes hello >/tmp/fifo100500 & head </tmp/fifo100500
[1] 12678
hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
hello
[1]+  Broken pipe: 13         yes hello > /tmp/fifo100500
```

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

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

Система межпроцессного взаимодействия IPC (Inter-Process Communication)

Система IPC (известная так же, как IPC System V) позволяет обеспечивать взаимодействие произвольных процессов в пределах локальной машины. Для этого она предоставляет взаимодействующим процессам возможность использования **общих**, или **разделяемых**, ресурсов:

- Очередь сообщений
- Массив семафоров
- Общая, или разделяемая, память

IPC. Механизм именования ресурсов

При создании ресурса с ним ассоциируется **ключ** — целочисленное значение.

Во избежание коллизий система предлагает некоторую унификацию именования IPC-ресурсов. Для генерации уникальных ключей в системе имеется библиотечная функция `ftok()`.

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

key_t ftok(char *filename, char proj);
```

IPC. Получение доступа к ресурсу

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

#include <sys/msg.h>
#include <sys/shm.h>
#include <sys/sem.h>


int msgget(key_t key,             int msgflags);
int shmget(key_t key, int size,   int shmflags);
int semget(key_t key, int nsems, int semflags);

// flags
IPC_PRIVATE // create private resource
IPC_CREAT   // create resource if not exists
IPC_EXCL    // only new

// errors
ENOENT // ресурс не существует, и не указан флаг IPC_CREAT;
EEXIST // ресурс существует, и установлены флаги IPC_CREAT | IPC_EXCL;
EACCES // не хватает прав доступа на подключение.
```

IPC. Очередь сообщений

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

#include <sys/msg.h>


int     msgsnd(int msqid, const void* msgp, size_t msgsz,              int msgflg);
ssize_t msgrcv(int msqid,       void* msgp, size_t msgsz, long msgtyp, int msgflg);
int     msgctl(int msqid, int cmd /* IPC_RMID */, struct msqid_ds* buf /* NULL */);


struct msgbuf {
    long msgtype;       /* message type, must be > 0 */
    /* message data */
};


IPC_NOWAIT  // non-blocking mode


// msgtyp == 0: the first message in the queue is read
// msgtyp >  0: the first message in the queue of type msgtyp is read
// msgtyp <  0: the first message in the queue with the lowest type less than or equal to
//              the absolute value of msgtyp will be read
```

IPC. Очередь сообщений

```
struct msgbuf {
    long    type = 1;
    char    buf[10];
    size_t current_len = 0;
};

int main(int argc, char** argv) {
    assert(argc == 3);

    srandomdev();

    key_t key = ftok(argv[1], 0);
    if (key == -1) {
        perror("failed to create key");
        return -1;
    }

    bool should_start = argv[2][0] == '1';

    int msgid = msgget(key, IPC_CREAT | 0600);
    if (msgid == -1) {
        perror("failed to get msgq");
        return -1;
    }

    struct msgbuf msg;
    while (true) {
        if (!should_start) {
            ssize_t size = msgrcv(msgid, &msg, sizeof(msg),
                                  msg.type == 1 ? 2 : 1, 0);
            if (size == -1) {
                if (errno == EIDRM)
                    return 0;
                perror("failed to rcv msg"); return -1;
            }
        }
        should_start = false;

        char letter = 'A' + random() % 26;
        printf("%d: %c\n", getpid(), letter);
        msg.buf[msg.current_len++] = letter;

        if (msg.current_len == sizeof(msg.buf)) {
            printf("done: %.*s\n", sizeof(msg.buf), msg.buf);
            if (msgctl(msgid, IPC_RMID, NULL)) {
                perror("failed to remove msgq"); return -1;
            }
            return 0;
        }

        msg.type = (msg.type == 1) ? 2 : 1;
        if (msgsnd(msgid, &msg, sizeof(msg), 0)) {
            perror("failed to send message"); return -1;
        }
    }
}
```

IPC. Очередь сообщений

```
[i.anferov@hostname ~]$ touch -m 644 /tmp/msgqueue100500
[i.anferov@hostname ~]$ ./a.out /tmp/msgqueue100500 1 & ./a.out /tmp/msgqueue100500 0
[1] 15415
15415: K
15416: I
15415: G
15416: Y
15415: M
15416: N
15415: L
15416: Z
15415: Z
15416: Y
buffer filled: KIGYMNLZZY
[1]+  Done                    ./a.out /tmp/msgqueue100500 1
```

IPC. Разделяемая память

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

#include <sys/shm.h>


int   shmget(key_t key, size_t size, int shmflg);
void* shmat (int shmid, const void *shmaddr, int shmflg);
int   shmdt (const void *shmaddr);
int   shmctl(int shmid, int cmd /* IPC_RMID */, struct shmid_ds* buf /* NULL */);


SHM_RND     // round the address down to a multiple of SHMLBA bytes (defined in <sys/shm.h>)
SHM_RDONLY  // read-only mapping
```

IPC. Массив семафоров

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

struct sembuf {
    u_short sem_num; /* semaphore # */
    short   sem_op;  /* semaphore operation:
                        sem_op > 0 – increase semaphore value,
                        sem_op < 0 – wait until semaphore value >= sem_op and decrease it,
                        sem_op == 0 – wait until semaphore value == 0 */
    short   sem_flg; /* IPC_NOWAIT | SEM_UNDO */
};

union semun {
    int     val;     /* value for SETVAL */
    u_short *array;  /* array for GETALL & SETALL */
};

int semget(key_t key, int nsems, int semflag);
int semop (int semid, struct sembuf* semop, size_t nops);
int semctl(int semid, int semnum, int cmd, ... /* optional union semun */);

IPC_RMID    // Immediately removes the semaphore set from the system
GETVAL      // Return the value of semaphore number semnum
SETVAL      // Set the value of semaphore number semnum to arg.val
GETPID      // Return the pid of the last process to perform an operation on semaphore number semnum
GETNCNT     // Return the number of processes waiting for semaphore semnum to increase its value
GETZCNT     // Return the number of processes waiting for semaphore semnum's value to become 0
GETALL      // Fetch the value of all of the semaphores in the set into arg.array
SETALL      // Set the values of all of the semaphores in the set to the values in arg.array
```

```
class Queue {
    const size_t capacity;
    size_t size = 0;
    size_t write_pos = 0;
    size_t read_pos = 0;
    int    data[];

public:
    Queue(size_t capacity): capacity(capacity) {}

    void push(int elem) {
        assert(size++ < capacity);
        data[write_pos++] = elem;
        if (write_pos == capacity)
            write_pos = 0;
    }

    int pop() {
        assert(size-- > 0);
        int res = data[read_pos++];
        if (read_pos == capacity)
            read_pos = 0;
        return res;
    }

    static size_t getRequiredMemSize(size_t capacity) {
        return sizeof(Queue) + sizeof(int) * capacity;
    }
};
```

```
enum SemType {
    QUEUE_LOCK = 0,
    POP_AVAILABLE,
    PUSH_AVAILABLE,
    TOTAL
};

class SharedQueue {
    int shmid;
    int semid;
    Queue* queue;
public:
    SharedQueue(key_t key):
        shmid(shmget(key, 0, 0)),
        semid(semget(key, SemType::TOTAL, 0))
    {
        if (shmid == -1) {
            perror("failed to open shm"); exit(1);
        }
        if (semid == -1) {
            perror("failed to open sem"); exit(1);
        }
        queue = (Queue*)shmat(shmid, NULL, 0);
        if (queue == (void*)-1) {
            perror("failed to open shm"); exit(1);
        }
    }
    void push(int elem);
    int pop();
};
```

```
void SharedQueue::push(int elem) {
    static struct sembuf lock_ops[] = {
        {
            .sem_num = SemType::QUEUE_LOCK,
            .sem_op = -1,
        },
        {
            .sem_num = SemType::PUSH_AVAILABLE,
            .sem_op = -1,
        },
    };
    if (semop(semid, lock_ops, sizeof(lock_ops) / sizeof(lock_ops[0]))) {
        perror("failed to lock semaphore to push new element"); exit(1);
    }

    queue->push(elem);

    static struct sembuf unlock_ops[] = {
        {
            .sem_num = SemType::QUEUE_LOCK,
            .sem_op = 1,
        },
        {
            .sem_num = SemType::POP_AVAILABLE,
            .sem_op = 1,
        },
    };
    if (semop(semid, unlock_ops, sizeof(unlock_ops) / sizeof(unlock_ops[0]))) {
        perror("failed to unlock semaphore after pushing new element"); exit(1);
    }
}
```

```
int SharedQueue::pop() {
    static struct sembuf lock_ops[] = {
        {
            .sem_num = SemType::QUEUE_LOCK,
            .sem_op = -1,
        },
        {
            .sem_num = SemType::POP_AVAILABLE,
            .sem_op = -1,
        },
    };
    if (semop(semid, lock_ops, sizeof(lock_ops) / sizeof(lock_ops[0]))) {
        perror("failed to lock semaphore to pop element"); exit(1);
    }

    int res = queue->pop();

    static struct sembuf unlock_ops[] = {
        {
            .sem_num = SemType::QUEUE_LOCK,
            .sem_op = 1,
        },
        {
            .sem_num = SemType::PUSH_AVAILABLE,
            .sem_op = 1,
        },
    };
    if (semop(semid, unlock_ops, sizeof(unlock_ops) / sizeof(unlock_ops[0]))) {
        perror("failed to unlock semaphore after popping element"); exit(1);
    }
    return res;
}
```

```cpp
int main(int argc, char** argv) {
    assert(argc == 3);

    key_t key = ftok(argv[1], 0);
    if (key == -1) {
        perror("failed to create key"); return -1;
    }

    size_t capacity = std::stoull(argv[2]);

    int shmid = shmget(key, Queue::getRequiredMemSize(capacity), IPC_CREAT | IPC_EXCL | 0600);
    if (shmid == -1) {
        perror("failed to create shared memory region"); return -1;
    }

    void* queue_mem = shmat(shmid, NULL, 0);
    new(queue_mem) Queue(capacity);

    int semid = semget(key, SemType::TOTAL, IPC_CREAT | IPC_EXCL | 0600);
    if (semid == -1) {
        perror("failed to create sem array"); return -1;
    }

    u_short init[] = {
        [SemType::QUEUE_LOCK] = 1,
        [SemType::POP_AVAILABLE] = 0,
        [SemType::PUSH_AVAILABLE] = (u_short)capacity,
    };
    semctl(semid, 0, SETALL, (union semun){ .array = init });
}
```

```
int main(int argc, char** argv) {
    assert(argc == 3);

    srandomdev();

    key_t key = ftok(argv[1], 0);
    if (key == -1) {
        perror("failed to create key"); return -1;
    }

    SharedQueue queue(key);

    bool producer = !strcmp(argv[2], "producer");

    for (int i = 0; i < 5; ++i) {
        if (producer) {
            int n = (int)random();
            queue.push(n);
            printf("%d: produced %d\n", getpid(), n);
        } else {
            int n = queue.pop();
            printf("%d: consumed %d\n", getpid(), n);
        }
    }
}
```

```
$ touch -m 644 /tmp/sharedqueue123
$ ./create_shared_queue /tmp/sharedqueue123 5
$ \
> for role in producer consumer; do
>     for i in {1..3}; do
>         ./use_shared_queue /tmp/sharedqueue123 $role &
>     done
> done
```

```
18852: produced 209214663
18852: produced 1038616888
18852: produced 1461195765
18852: produced 1282151790
18852: produced 1766617917
18854: consumed 209214663
18854: consumed 1038616888
18854: consumed 1461195765
18854: consumed 1282151790
18851: produced 1184647616
18854: consumed 1766617917
18851: produced 1575360679
18851: produced 822665796
18851: produced 1014047724
18853: produced 502704283
18855: consumed 1184647616
18851: produced 345742988
18855: consumed 502704283
18855: consumed 1575360679
18853: produced 2118629247
18855: consumed 822665796
18853: produced 651276396
18855: consumed 1014047724
18853: produced 50274542
18853: produced 1571117976
18856: consumed 345742988
18856: consumed 2118629247
18856: consumed 651276396
18856: consumed 50274542
18856: consumed 1571117976
```

Socket API. Типы сокетов

По типу передаваемых данных:

- Соединение с использованием виртуального канала
- Передача сообщений (датаграмм) без установления соединения

По типу коммуникационного домена:

- Взаимодействие внутри одной операционной системы (домен `AF_UNIX`)
  - Адрес — любое допустимое имя файла
- Сетевое взаимодействие (домен `AF_INET`)
  - Адрес — имя хоста + номер порта

Socket API

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

int socket(int domain, int type, int protocol /* 0 */);

AF_UNIX  // local socket domain
AF_INET  // network socket domain

SOCK_STREAM  // stream socket type
SOCK_DGRAM   // datagram socket type
```

Socket API. Связывание сокета с адресом

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

int bind(int socket, const struct sockaddr *address, socklen_t address_len);


#include <sys/un.h>

struct sockaddr_un {
    sa_family_t sun_family;    /* AF_UNIX */
    char        sun_path[108]; /* Pathname */
};
socklen_t SUN_LEN(struct sockaddr_un* address);


#include <netinet/in.h>

struct in_addr {
    in_addr_t s_addr;
};

struct sockaddr_in {
    short sin_family;  /* == AF_INET */
    u_short sin_port;  /* port number */
    struct in_addr sin_addr;  /* host IP address */
    char sin_zero[8];  /* not used */
};
```

Socket API. Установление соединения

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


// client
int connect(int sockfd, const struct sockaddr *server_address, socklen_t server_address_len);


// server
int listen (int sockfd, int backlog);
int accept (int sockfd, struct sockaddr *client_addr, int *client_addr_len);
```

Socket API. Передача данных

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


// SOCK_STREAM
ssize_t write   (int sockfd, const void* buffer, size_t length);
ssize_t read    (int sockfd,       void* buffer, size_t length);


ssize_t send    (int sockfd, const void* buffer, size_t length, int flags /* MSG_OOB */);
ssize_t recv    (int sockfd,       void* buffer, size_t length, int flags /* MSG_OOB | MSG_PEEK | MSG_WAITALL */);


// SOCK_DGRAM
ssize_t sendto  (int sockfd, const void* buffer, size_t length, int flags,
                                                                const struct sockaddr *dst_addr, socklen_t  dst_addr_len);

ssize_t recvfrom(int sockfd,       void* buffer, size_t length, int flags, struct sockaddr *src_addr, socklen_t* src_addr_len);
```

Socket API. Закрытие соединения

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


int shutdown(int sockfd, int how /* SHUT_RD, SHUT_WR, SHUT_RDWR */);
int close   (int sockfd);
```

```c
#define EXIT(condition, reason) \
    if (condition) { if (errno) perror(reason); else fprintf(stderr, "%s\n", reason); exit(1); }

int main(int argc, char** argv) {
    EXIT(argc != 2, "path to socket file is expected");
    int fd = socket(AF_UNIX, SOCK_STREAM, 0);
    EXIT(fd == -1, "failed to create socket");
    struct sockaddr_un addr = { .sun_family = AF_UNIX };
    snprintf(addr.sun_path, sizeof(addr.sun_path), "%s", argv[1]);
    unlink(argv[1]);
    EXIT(bind(fd, (struct sockaddr*)&addr, (socklen_t)SUN_LEN(&addr)), "failed to bind socket");
    EXIT(listen(fd, 1024), "failed to start listening socket");

    char buf[1024];
    while (true) {
        int client_fd = accept(fd, NULL, NULL);
        EXIT(client_fd == -1, "failed to accept new connection");


        while (true) {
            ssize_t r = read(client_fd, buf, sizeof(buf));
            EXIT(r == -1, "failed to read data from client");
            if (r == 0) {
                close(client_fd);
                break;
            }
            ssize_t w = write(client_fd, buf, r);
            EXIT(w != r, "failed to write data to client socket");
        }
    }
}
```

```c
#define EXIT(condition, reason) \
    if (condition) { if (errno) perror(reason); else fprintf(stderr, "%s\n", reason); exit(1); }


int main(int argc, char** argv) {
    EXIT(argc != 2, "path to socket file is expected");
    int fd = socket(AF_UNIX, SOCK_STREAM, 0);
    EXIT(fd == -1, "failed to create socket");
    struct sockaddr_un addr = { .sun_family = AF_UNIX };
    snprintf(addr.sun_path, sizeof(addr.sun_path), "%s", argv[1]);
    EXIT(connect(fd, (struct sockaddr*)&addr, (socklen_t)SUN_LEN(&addr)), "failed to connect to socket");

    char buf[1024];

    int pid = fork();
    EXIT(pid == -1, "fork failed");

    while (true) {
        ssize_t r = read(pid ? 0 : fd, buf, sizeof(buf));
        EXIT(r == -1, "failed to read data");
        if (r == 0)
            break;
        ssize_t w = write(pid ? fd : 1, buf, r);
        EXIT(w != r, "failed to write data");
    }
}
```

```
#define EXIT(condition, reason) \
    if (condition) { if (errno) perror(reason); else fprintf(stderr, "%s\n", reason); exit(1); }

int main(int argc, char** argv) {
    EXIT(argc != 3, "ip and port are expected");
    int fd = socket(AF_INET, SOCK_STREAM, 0);
    EXIT(fd == -1, "failed to create socket");
    struct sockaddr_in addr = { .sin_family = AF_INET };
    EXIT(inet_aton(argv[1], &addr.sin_addr) != 1, "failed to parse IP address");
    addr.sin_port = htons(std::stoi(argv[2]));
    EXIT(bind(fd, (struct sockaddr*)&addr, (socklen_t)sizeof(addr)), "failed to bind socket");
    EXIT(listen(fd, 1024), "failed to start listening socket");

    char buf[1024];
    while (true) {
        int client_fd = accept(fd, NULL, NULL);
        EXIT(client_fd == -1, "failed to accept new connection");

        while (true) {
            ssize_t r = read(client_fd, buf, sizeof(buf));
            EXIT(r == -1, "failed to read data from client");
            if (r == 0) {
                close(client_fd);
                break;
            }
            ssize_t w = write(client_fd, buf, r);
            EXIT(w != r, "failed to write data to client socket");
        }
    }
}
```

```
#define EXIT(condition, reason) \
    if (condition) { if (errno) perror(reason); else fprintf(stderr, "%s\n", reason); exit(1); }


int main(int argc, char** argv) {
    EXIT(argc != 3, "ip and port are expected");
    int fd = socket(AF_INET, SOCK_STREAM, 0);
    EXIT(fd == -1, "failed to create socket");
    struct sockaddr_in addr = { .sin_family = AF_INET };
    EXIT(inet_aton(argv[1], &addr.sin_addr) != 1, "failed to parse IP address");
    addr.sin_port = htons(std::stoi(argv[2]));
    EXIT(connect(fd, (struct sockaddr*)&addr, (socklen_t)sizeof(addr)), "failed to connect to socket");
    char buf[1024];

    int pid = fork();
    EXIT(pid == -1, "fork failed");

    while (true) {
        ssize_t r = read(pid ? 0 : fd, buf, sizeof(buf));
        EXIT(r == -1, "failed to read data");
        if (r == 0)
            break;
        ssize_t w = write(pid ? fd : 1, buf, r);
        EXIT(w != r, "failed to write data");
    }
}
```

Перевод файлового дескриптора в неблокирующий режим

```
#include <fcntl.h>


int fcntl(int fd, int cmd, ...);

F_GETFL  // get fd flags
F_SETFL  // set fd flags

O_NONBLOCK  // don't block on IO-operations on fd



int flags = fcntl(fd, F_GETFL);
fcntl(fd, F_SETFL, flags | O_NONBLOCK);



ssize_t r = read(fd, buf, sizeof(buf));
if (r == -1 && errno == EWOULDBLOCK) {
    ...
}
```

Ожидание событий на нескольких файловых дескрипторах

```
#include <sys/select.h>


void FD_CLR  (int fd, fd_set* fdset);
void FD_COPY (const fd_set* fdset_orig, fd_set* fdset_copy);
int  FD_ISSET(int fd, const fd_set* fdset);
void FD_SET  (int fd, fd_set* fdset);
void FD_ZERO (fd_set* fdset);


int select(int nfds /* max fd + 1 */, fd_set* rfds, fd_set* wfds, fd_set* efds, struct timeval* timeout);
```

Спасибо за внимание!

E-mail: i.anferov@corp.mail.ru
Telegram: igor_anferov
GitHub: igor-anferov

Игорь Анфёров

образование