PIPE, SIGNAL

```c
int flag = 0;
void handler(int sig_numb)
{
    flag = 1;
    printf(" Signal handler %d installed.\n", sig_numb);
}
int main(void)
{
    pid_t chpids[2];
    int wstatus;
    pid_t w;
    char *texts[2] = {"xx", "y-y-y-y-y"};
    int pipefd[2];
    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    if (signal(SIGINT, handler) == SIG_ERR) {
        perror("Ошибка signal");
        exit(EXIT_FAILURE);
    }
    printf("Push Ctrl+C, to send message from childs.\n");
    sleep(2);
    for (size_t i = 0; i < 2; ++i) {
        if ((chpids[i] = fork()) == -1) {
            perror("Can not fork.\n");
            exit(EXIT_FAILURE);
        }
        else if (chpids[i] == 0) {

            if (flag) {
                printf("PID = %d write: '%s'\n", getpid(), texts[i]);
                close(pipefd[0]);
                write(pipefd[1], texts[i], strlen(texts[i]));
            } else
                printf("PID = %d no signal \n", getpid());

            exit(EXIT_SUCCESS);
        }
    }
    for (size_t i = 0; i < 2; ++i) {
        w = wait(&wstatus);
        if (w == -1) {
            perror("waitpid");
            exit(EXIT_FAILURE);
        }
        printf("terminated pid = %d\t", w);
        if (WIFEXITED(wstatus)) {
            printf("exited, status = %d\n", WEXITSTATUS(wstatus));
        } else if (WIFSIGNALED(wstatus)) {
            printf("killed by signal %d\n", WTERMSIG(wstatus));
        }
    }
```

```
    char buf[50];
    for (size_t i = 0; i < 2; ++i) {
        close(pipefd[1]);
        read(pipefd[0], &buf, strlen(texts[i]));
        printf("Received message from PID = %d: %s\n", chpids[i], buf);
        buf[0] = 0;
    }
    close(pipefd[1]);
    read(pipefd[0], buf, 1);
    printf("Received message: %s\n", buf);
    exit(EXIT_SUCCESS);
}
```

DAEMON

```c
#include "apue.h"
#include <syslog.h>
#include <fcntl.h>
#include <sys/resource.h>
#include <errno.h>
#include <pthread.h>
#define LOCKFILE "/var/run/daemon.pid"
#define CONFFILE "/etc/daemon.conf"
#define LOCKMODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
// Инициализация процесса-демона
void daemonize(const char *cmd)
{
    int                 i, fd0, fd1, fd2;
    pid_t               pid;
    struct rlimit       rl;
    struct sigaction    sa;
    // Сброс маску создания файлов в значение 0
    umask(0);
    // Получить максимально возможный номер дескриптора файла.
    if (getrlimit(RLIMIT_NOFILE, &rl) == -1)
        err_quit("%s: невозможно получить максимальный номер дескриптора ",
cmd);
    // Стать лидером нового сеанса, чтобы утратить управляющий терминал
    if ((pid = fork()) == -1)
        err_quit("%s: ошибка вызова функции fork", cmd);
    else if (pid != 0)  // родительский процесс
        exit(0);
    // Создаем новый сеанс
    if (setsid() == -1) {
        perror("Ошибка setsid");
        exit(EXIT_FAILURE);
    }
    // Обеспечить невозможность обретения управляющего терминала в будущем.
    sa.sa_handler = SIG_IGN;
    sigemptyset(&sa.sa_mask);   // количество различных сигналов может
превышать количество бит в целочисленном типе, поэтому в большинстве случаев
нельзя использовать тип int
    sa.sa_flags = 0;
    if (sigaction(SIGHUP, &sa, NULL) == -1)
        err_quit("%s:  невозможно игнорировать сигнал SIGHUP", cmd);
    // Назначить корневой каталог текущим рабочим каталогом,
    // чтобы впоследствии можно было отмонтировать файловую систему.
    if (chdir("/") == -1)
        err_quit("%s:  невозможно сделать текущим рабочим каталогом  /", cmd);
    // Закрыть все открытые файловые дескрипторы.
    if (rl.rlim_max == RLIM_INFINITY)
        rl.rlim_max = 1024; // макс число файловых дескрипторов которые могут
быть открыты
    for (i = 0; i < rl.rlim_max; i++)
        close(i);
    // Присоединить файловые дескрипторы 0, 1 и 2 к /dev/null.
    fd0 = open("/dev/null", O_RDWR);
    fd1 = dup(0);
```

```
    fd2 = dup(0);
    // Инициализировать файл журнала.
    openlog(cmd, LOG_CONS, LOG_DAEMON);
    if (fd0 != 0 || fd1 != 1 || fd2 != 2){
        syslog(LOG_ERR, "ошибочные файловые дескрипторы %d %d %d",
               fd0, fd1, fd2);
        exit(1);
    }}
// Установка блокировки для записи на весь файл
int lockfile(int fd)
{
    struct flock fl;
    fl.l_type = F_WRLCK;
    fl.l_start = 0;
    fl.l_whence = SEEK_SET;
    fl.l_len = 0;
    return (fcntl(fd, F_SETLK, &fl));
}
// Функция, которая гарантирует запуск только одной копии демона
int already_running(void)
{
    int fd;
    int rc;
    int conffd;
    char buf[16];
    char conf_buf[256];
    int perms = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
    fd = open(LOCKFILE, O_RDWR | O_CREAT, perms);
    if (fd == -1) {
        syslog(LOG_ERR, "невозможно открыть %s: %s", LOCKFILE,
strerror(errno));
        exit(1);
    }
    if (lockfile(fd) == -1) {
        if (errno == EACCES || errno == EAGAIN)
        {
            close(fd);
            return 1;
        }
        syslog(LOG_ERR, "невозможно установить блокировку на %s: %s", LOCKFILE,
strerror(errno));
        exit(1);
    }
    ftruncate(fd, 0);
    sprintf(buf, "%ld", (long)getpid());
    write(fd, buf, strlen(buf) + 1);
    if ((conffd = open(CONFFILE, O_RDWR | O_CREAT, perms)) == -1) // username
userid {
        syslog(LOG_ERR, "ошибка открытия файла %s", CONFFILE);
        exit(EXIT_FAILURE);
    }
    ftruncate(conffd, 0);
    sprintf(conf_buf, "%ld\n%s", (long) getuid(), getlogin());
    rc = write(conffd, conf_buf, strlen(conf_buf) + 1);
```

```
    if (rc == -1) {
        syslog(LOG_ERR, "ошибка записис в файл %s", CONFFILE);
        exit(EXIT_FAILURE);
    }
    close(conffd);
    return 0;
}
void reread(void) {
    FILE *fd;
    int pid,userid;
    char login[50];
    if ((fd = fopen(CONFFILE, "r")) == NULL) {
        syslog(LOG_INFO, "Ошибка fopen " CONFFILE "");
        exit(EXIT_FAILURE);
    }
    fscanf(fd, "%d\n", &userid);
    fscanf(fd, "%s\n", login);
    fclose(fd);
    syslog(LOG_INFO, "UserName: %s", login);
    syslog(LOG_INFO, "UserID: %d", userid);
}
void *thr_fn(void *arg) {
    sigset_t *mask = arg;
    int err, signo;
    for (;;)
    {
        err = sigwait(mask, &signo);
        if (err != 0)
        {
            syslog(LOG_ERR, "sigwait failed");
            exit(EXIT_FAILURE);
        }
        switch (signo){
        case SIGHUP:
            syslog(LOG_INFO, "Re-reading configuration file");
            reread();
            break;
        case SIGTERM:
            syslog(LOG_INFO, "got SIGTERM; exiting");
            if (remove(CONFFILE) == -1)
                syslog(LOG_ERR, "error remove %s", CONFFILE);
            exit(EXIT_SUCCESS);
        default:
            syslog(LOG_INFO, "unexpected signal %d\n", signo);
        }}}
int main(int argc, char *argv[]) {
    int err;
    pthread_t tid;
    char *cmd;
    struct sigaction sa;
    sigset_t mask;
    // cmd - имя исполняемого файла (без пути)
    if ((cmd = strrchr(argv[0], '/')) == NULL)
        cmd = argv[0];
```

```
    else
        cmd++;
    // Перейти в режим демона.
    daemonize(cmd);
    // Убедиться, что ранее не была запущена другая копия демона.
    if (already_running()) {
        syslog(LOG_ERR, "демон уже запущен");
        exit(EXIT_FAILURE);
    }
    // до этого момента SIGHUP игнорируется
    // Восстановить действие по умолчанию для сигнала SIGHUP
    sa.sa_handler = SIG_DFL;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGHUP, &sa, NULL) == -1)
        err_quit("%s:  невозможно восстановить действие SIG_DFL для SIGHUP");
    // и заблокировать все сигналы.
    sigfillset(&mask);
    if ((err = pthread_sigmask(SIG_BLOCK, &mask, NULL)) != 0)
        err_exit(err, "ошибка выполнения операции SIG_BLOCK");
    // Создать поток для обработки SIGHUP и SIGTERM.
    err = pthread_create(&tid, NULL, thr_fn, &mask);
    if (err == -1)
        err_exit(err, "can't create thread");
    while (1)
    {
        time_t cur_time = time(NULL);
        syslog(LOG_NOTICE, "Time: %s", ctime(&cur_time));
        sleep(10);
    }
    return 0;
}
```

```
struct sembuf Pprod[2] = {{BEMPTY, P, SEM_UNDO}, {BINARY, P, SEM_UNDO}};
struct sembuf Vprod[2] = {{BINARY, V, SEM_UNDO}, {BFULL, V, SEM_UNDO}};
struct sembuf Pcons[2] = {{BFULL, P, SEM_UNDO}, {BINARY, P, SEM_UNDO}};
struct sembuf Vcons[2] = {{BINARY, V, SEM_UNDO}, {BEMPTY, V, SEM_UNDO}};

producer:
        if (semop(semid, Pprod, 2) == -1) {
            char err_msg[100];
            sprintf(err_msg, "ERR: semop PID=%d, errno=%d (EINTR=%d)",
getpid(), errno, EINTR);
            errExit(err_msg);
        }
        printf("Производитель PID=%d положил '%c'\n", getpid(), *(*prod_ptr));
        if (semop(semid, Vprod, 2) == -1)
        errExit("semop");

consumer
        if (semop(semid, Pcons, 2) == -1) {
            char err_msg[100];
            sprintf(err_msg, "ERR: semop PID=%d, errno=%d (EINTR=%d)",
getpid(), errno, EINTR);
            errExit(err_msg);
        }
        printf("Потребитель   PID=%d взял    '%c'\n", getpid(), *(*cons_ptr));
        if (semop(semid, Vcons, 2) == -1)
            errExit("semop");

main:
    if (signal(SIGINT, myhandler) == SIG_ERR)
        errExit("signal");
    key = ftok(argv[0], 1);
    if (key == -1)
        errExit("ftok");
    shmid = shmget(key, N_BUF + 3, IPC_CREAT | perms);
    if (shmid == -1)
        errExit("shmget");
    addr = shmat(shmid, NULL, 0);
    if (addr == (void *) -1)
        errExit("shmat");
    semid = semget(key, 3, IPC_CREAT | perms);
    if (semid == -1)
        errExit("semget");
    if (semctl(semid, BINARY, SETVAL, 1) == -1)
        errExit("semctl");
    if (semctl(semid, BEMPTY, SETVAL, N_BUF) == -1)
        errExit("semctl");
    if (semctl(semid, BFULL, SETVAL, 0) == -1)
        errExit("semctl");
```

WR

```
struct sembuf start_read[] = {
    {readers_queue, 1, 0},
    {active_writer, 0, 0},
    {writers_queue, 0, 0},
    {numb_of_readers, 1, 0},
    {readers_queue, -1, 0}
};
struct sembuf stop_read[] = { {numb_of_readers, -1, 0} };
struct sembuf start_write[] = {
    {writers_queue, 1, 0},
    {numb_of_readers, 0, 0},
    {bin_sem, -1, 0},
    {active_writer, 1, 0},
    {writers_queue, -1, 0}
};
struct sembuf stop_write[] = { {active_writer, -1, 0}, {bin_sem, 1, 0} };
```

READER

```
        if (semop(semid, start_read, 5) == -1) {
            exit(EXIT_FAILURE);
        }
        printf("Читатель PID=%d считал '%c'\n", getpid(), *buf);
        if (semop(semid, stop_read, 1) == -1) {
            exit(EXIT_FAILURE);
        }
```

WRITER

```
        if (semop(semid, start_write, 5) == -1) {
            exit(EXIT_FAILURE);
        }
        printf("Писатель PID=%d записал '%c'\n", getpid(), *buf);
        if (semop(semid, stop_write, 2) == -1){
            exit(EXIT_FAILURE);
        }
```

WINDOWS

```
void start_read() {
    InterlockedIncrement(&readers_queue);
    if (active_writer || writers_queue)
        WaitForSingleObject(can_read, INFINITE);
    WaitForSingleObject(mutex, INFINITE);
    InterlockedDecrement(&readers_queue);
    InterlockedIncrement(&active_readers);
    SetEvent(can_read);
    ReleaseMutex(mutex);
}

void stop_read() {
    InterlockedDecrement(&active_readers);
    if (active_readers == 0)
        SetEvent(can_write);
}

void start_write() {
    InterlockedIncrement(&writers_queue);
    if (active_readers || active_writer)
        WaitForSingleObject(can_write, INFINITE);
    InterlockedDecrement(&writers_queue);
    InterlockedIncrement(&active_writer);
}

void stop_write() {
    InterlockedDecrement(&active_writer);
    ResetEvent(can_write);
    if (readers_queue > 0)
        SetEvent(can_read);
    else
        SetEvent(can_write);
}
```