

Лекция №4

# Многопоточное программирование

Дмитрий Калугин-Балашов

# Создание процессов



---

```
pid_t fork();
```

# Создание процессов



---

```
pid_t fork();
```

```
pid_t vfork();
```

# Создание процессов



---

```
pid_t fork();  
pid_t vfork();  
pid_t forkpty(int *amaster,  
              char *name,  
              const struct termios *termp,  
              const struct winsize *winp);
```

# Copy-on-write



# Copy-on-write

---



7

6

5

4

3

2

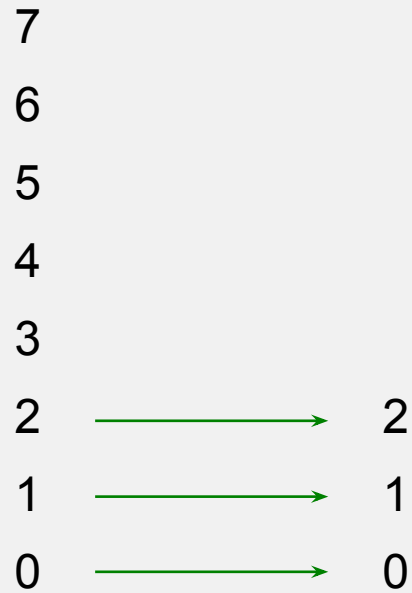
1

0

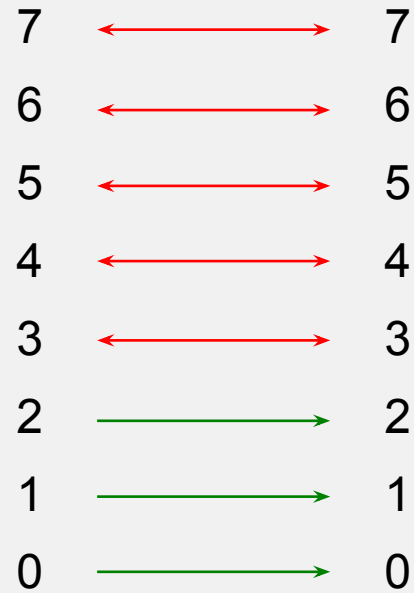


# Copy-on-write

---



# Copy-on-write





# Copy-on-write



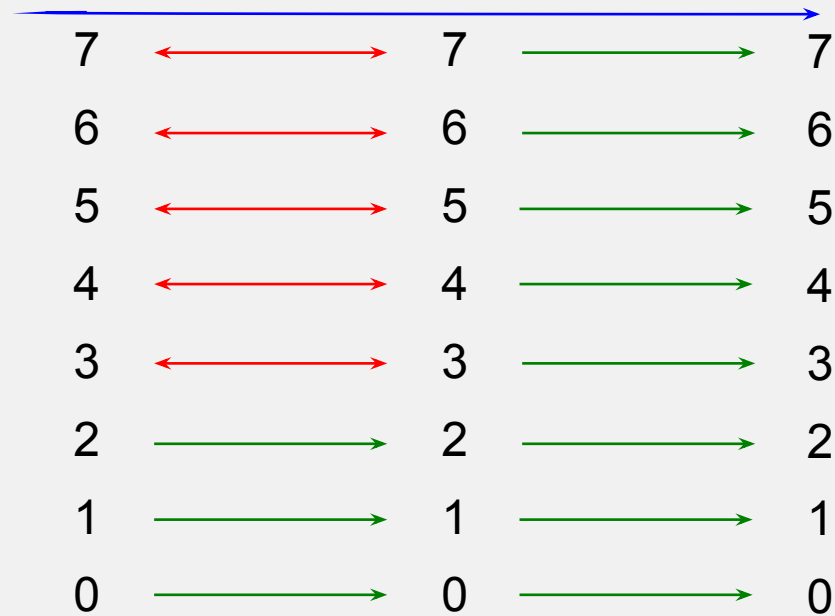
fork()



# Copy-on-write



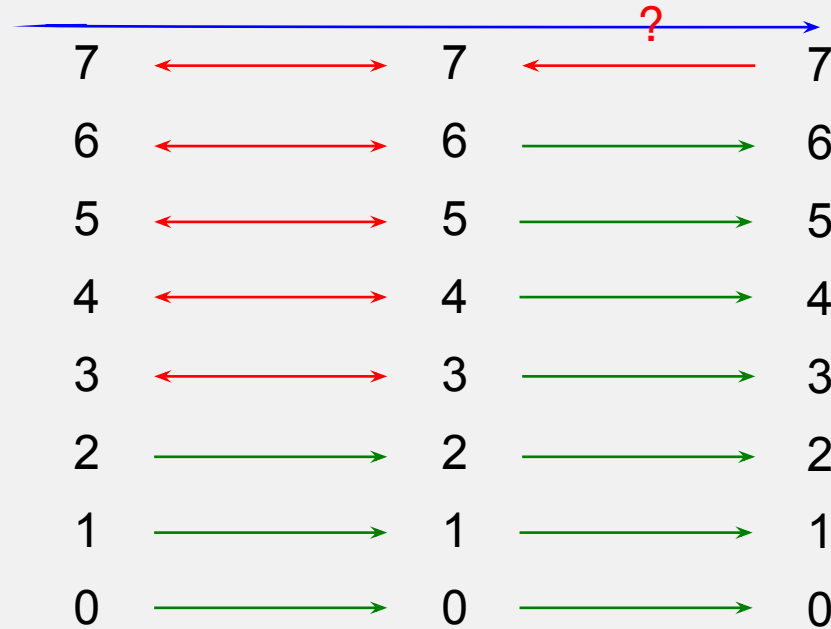
fork()



# Copy-on-write



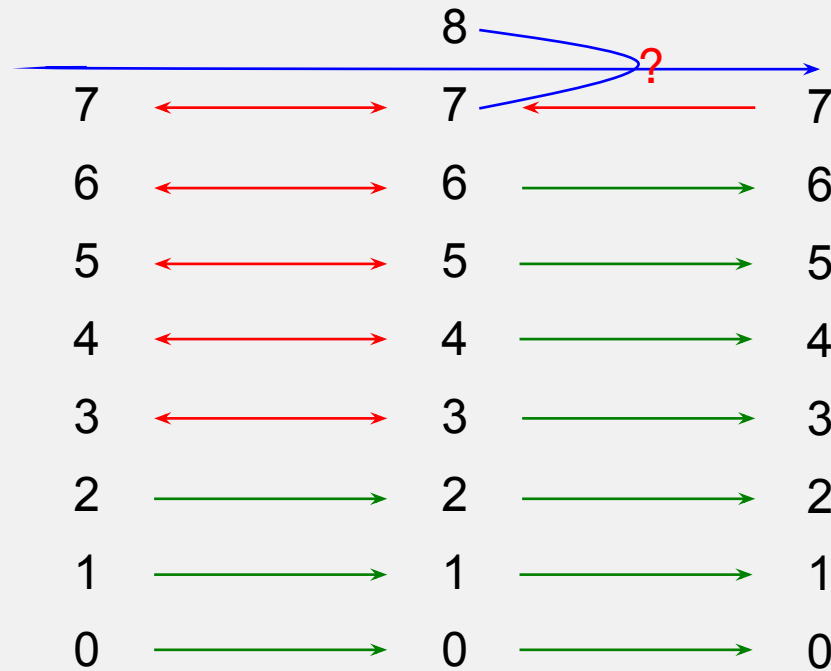
fork()



# Copy-on-write



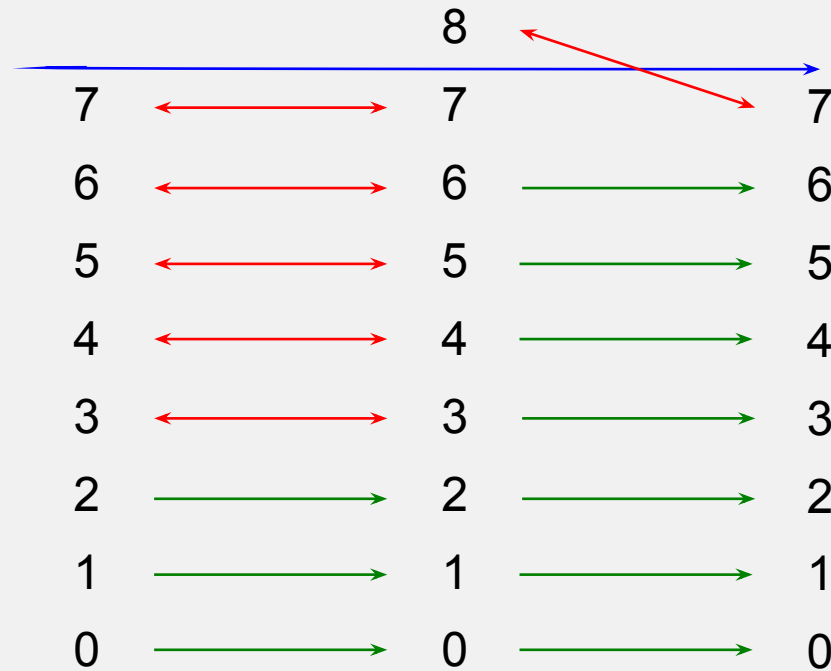
fork()



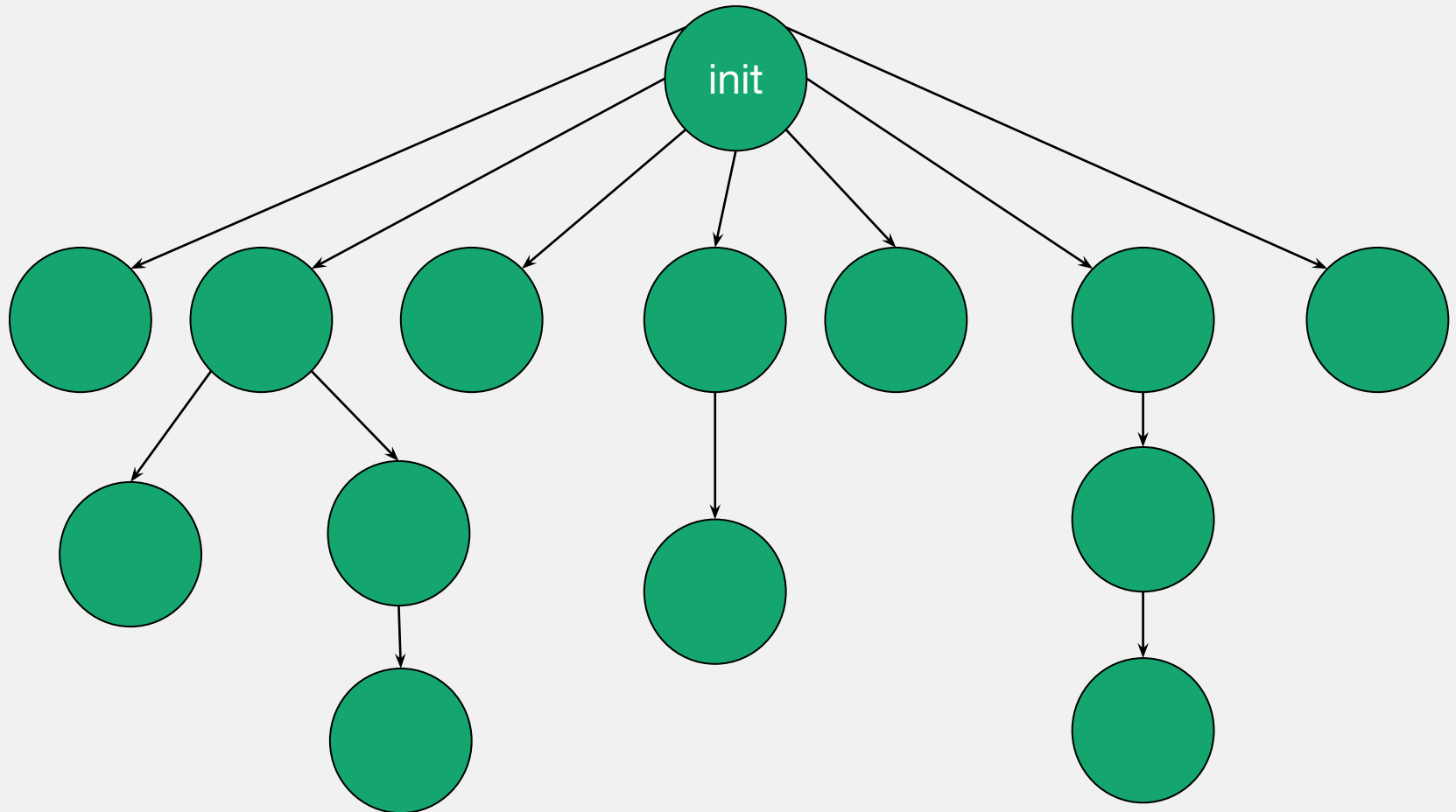
# Copy-on-write



fork()



# Дерево процессов





# Атрибуты процесса



## PID и PPID (Parent PID).

1. `pid_t getpid();`
2. `pid_t getppid();`



## PID и PPID (Parent PID).

```
1. pid_t getpid();
2. pid_t getppid();
3. // PID=1 для init.
4. // Если родительский процесс завершается,
5. // потомок получает PPID=1.
```





# Атрибуты процесса



## UID, GID, EUID, EGID.

```
1. // “Кто создал?”
2. // (Реальные идентификаторы пользователя и группы).

3. uid_t getuid();
4. int setuid(uid_t uid);
5. gid_t getgid();
6. int setgid(gid_t gid);

7. // “От чьего лица выполняется?”
8. // (Эффективные идентификаторы пользователя и группы).

9. uid_t geteuid();
10. int seteuid(uid_t uid);
11. gid_t getegid();
12. int setegid(gid_t gid);
```



## Корневой каталог.

```
1.  int chroot(const char *path);  
2.  // --- /var/new_root/somefile.txt  
3.  // --- chroot("/var/new_root");  
4.  // --- /somefile.txt
```



## Рабочий каталог.

1. `int chdir(const char *path);`
2. `int fchdir(int fd);`



## Приоритет.

```
1.  int nice(int incr);  
  
2.  // NZERO = 20  
3.  // NICE = 0..39  
4.  // NICE-20
```



## Ограничения.

```
1.  int getrlimit(int resource, struct rlimit *rlp);
2.  int setrlimit(int resource, const struct rlimit *rlp);

3.  struct rlimit {
4.  rlim_t rlim_cur;
5.  rlim_t rlim_max;
6.  }

7.  // RLIMIT_CORE, RLIMIT_CPU, RLIMIT_DATA, RLIMIT_FSIZE,
   // RLIMIT_NOFILE, RLIMIT_STACK, RLIMIT_AS

8.  // RLIM_SAVED_MAX, RLIM_SAVED_CUR, RLIM_INFINITY
```



## Ограничения.

1. `long ulimit(int cmd, ...); // Устаревший`
2. `int getrusage(int who, struct rusage *r_usage);`
3. `// RUSAGE_SELF, RUSAGE_CHILDREN`
4. `int prlimit(pid_t pid,`
5. `int resource,`
6. `const struct rlimit *new_limit,`
7. `struct rlimit *old_limit);`



## Переменные окружения.

```
1.  extern char **environ;
2.  // Имя=Значение
3.  // Имя=Значение
4.  // ...
5.  // Имя=Значение
6.  // \0

7.  char *getenv(const char *var);
8.  int  putenv(char *string);
9.  int  setenv(const char *var, const char *val, int overwrite);
10. int  unsetenv(const char *var);
```



## Порождение процесса через `exec`.

- `int execl(const char *path, const char *arg, ...);`
- `int execv(const char *path, char *const argv[]);`
- `int execlp(const char *path, const char *arg, ..., char *const envp[]);`
- `int execve(const char *path, char *const argv[], char *const envp[]);`
- `int execlp(const char *file, const char *arg, ...);`
- `int execvp(const char *file, char *const argv[]);`





## Порождение процесса через exec.

- `int execl(const char *path, const char *arg, ...);`
- `int execv(const char *path, char *const argv[]);`
- `int execl(const char *path, const char *arg, ..., char * const envp[]);`
- `int execve(const char *path, char *const argv[], char *const envp[]);`
- `int execlp(const char *file, const char *arg, ...);`
- `int execvp(const char *file, char *const argv[]);`



## Порождение процесса через exec.

- `int execl(const char *path, const char *arg, ...);`
- `int execv(const char *path, char *const argv[]);`
- `int execl(const char *path, const char *arg, ..., char * const envp[]);`
- `int execve(const char *path, char *const argv[], char *const envp[]);`
- `int execlp(const char *file, const char *arg, ...);`
- `int execvp(const char *file, char *const argv[]);`



## Порождение процесса через exec.

- `int execl(const char *path, const char *arg, ...);`
- `int execv(const char *path, char *const argv[]);`
- `int execl(const char *path, const char *arg, ..., char * const envp[]);`
- `int execve(const char *path, char *const argv[], char *const envp[]);`
- `int execlp(const char *file, const char *arg, ...);`
- `int execvp(const char *file, char *const argv[]);`



## Порождение процесса через exec.

- `int execl(const char *path, const char *arg, ...);`
- `int execv(const char *path, char *const argv[]);`
- `int execl(const char *path, const char *arg, ..., char * const envp[]);`
- `int execve(const char *path, char *const argv[], char *const envp[]);`
- `int execlp(const char *file, const char *arg, ...);`
- `int execvp(const char *file, char *const argv[]);`



# Порождение процессов



## Порождение процесса через `system`.

- `int system(const char *command);`



## Как предотвратить зомби?

```
1. pid_t waitpid(pid_t pid, int *statusp, int options);  
2. // pid <- PID или -1  
3. // options <- WNOHANG  
4. // wait(&status) = waitpid(-1, &status, 0);
```

# Исполнение процесса

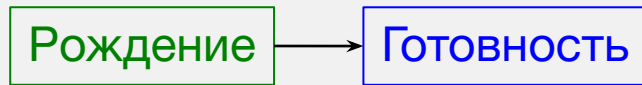
---



Рождение

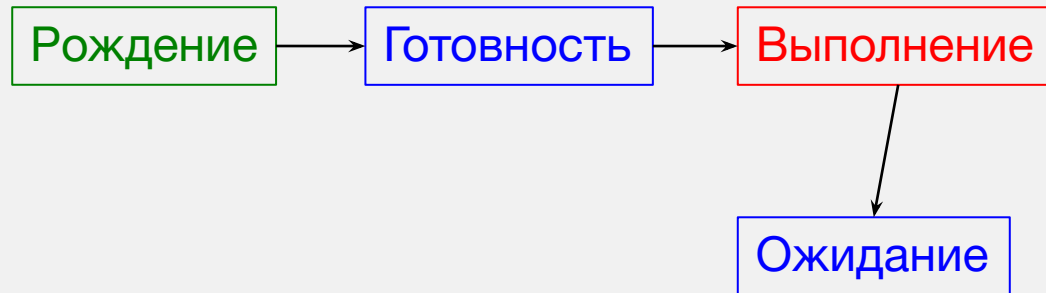
# Исполнение процесса

---

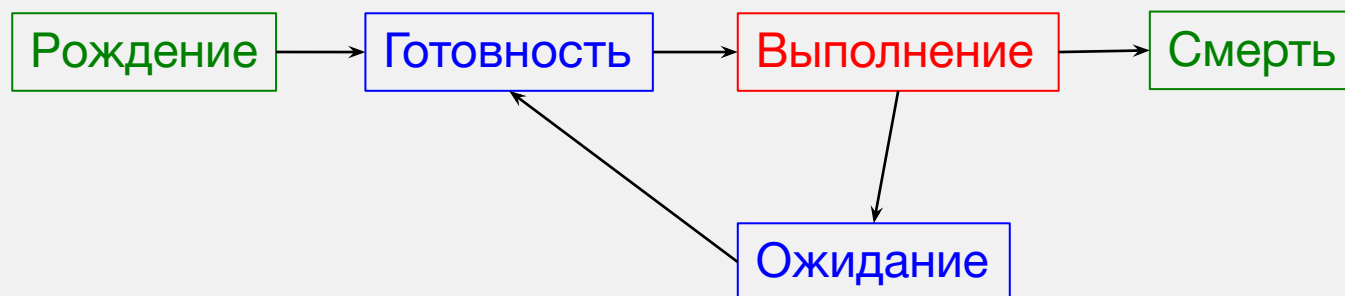




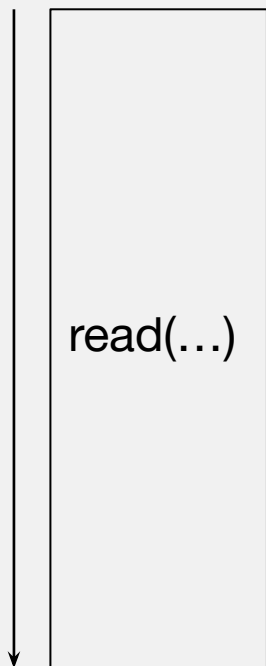
# Исполнение процесса



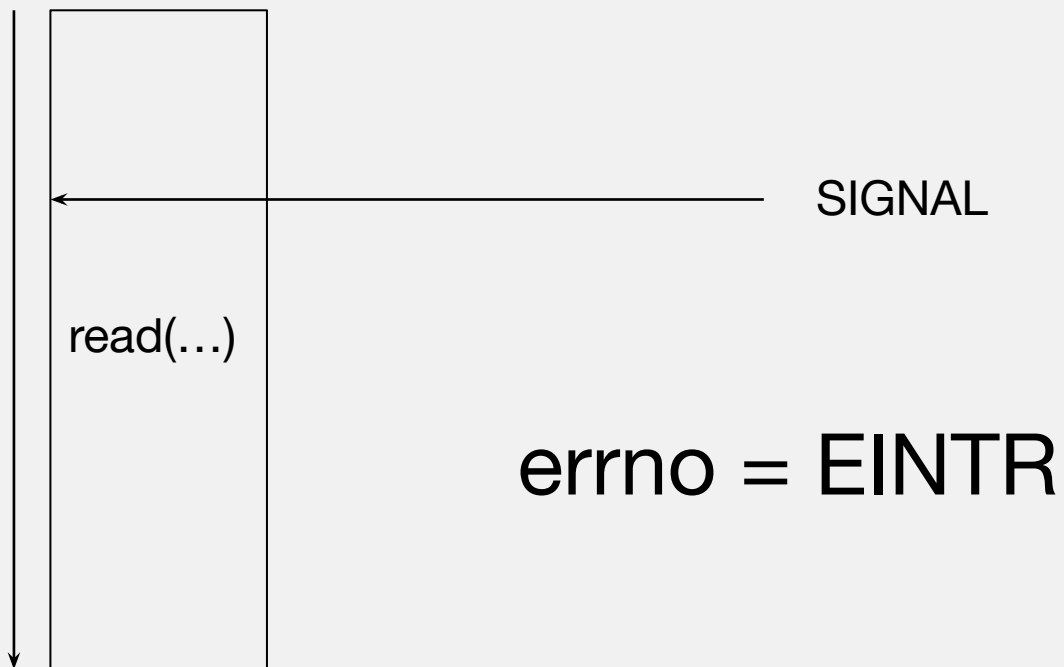
# Исполнение процесса



# Сигналы









## Основные типы сигналов.

1. `kill -l`
2. SIGABRT
3. SIGALRM
4. SIGCHLD
5. SIGINT
6. SIGTERM
7. SIGKILL
8. SIGUSR1
9. SIGUSR2
10. SIGHUP
11. SIGRTMIN
12. SIGRTMAX



## ANSI C.

```
1. void handler(int signum)
2. {
3.     // ...
4. }

5. // ANSI C
6. signal(SIGUSR1, handler);
7. signal(SIGUSR2, SIG_IGN);
8. signal(SIGTERM, SIG_DFL);
```



## POSIX.

```
1. // POSIX
2. struct sigaction {
3.     void (*sa_handler)(int);
4.     void (*sa_sigaction)(int, siginfo_t *, void *);
5.     sigset_t sa_mask;
6.     int sa_flags;
7.     void (*sa_restorer)(void);
8. }
```





## POSIX.

```
1. // POSIX
2. struct sigaction {
3.     void (*sa_handler)(int);
4.     void (*sa_sigaction)(int, siginfo_t *, void *);
5.     sigset_t sa_mask;
6.     int sa_flags;
7.     void (*sa_restorer)(void);
8. }
```



## POSIX.

```
1. // POSIX
2. struct sigaction {
3.     void (*sa_handler)(int);
4.     void (*sa_sigaction)(int, siginfo_t *, void *);
5.     sigset_t sa_mask;
6.     int sa_flags;
7.     void (*sa_restorer)(void);
8. }

9. int sigemptyset(sigset_t *set);
10. int sigfillset(sigset_t *set);
11. int sigaddset(sigset_t *set, int signum);
12. int sigdelset(sigset_t *set, int signum);
13. int sigismember(const sigset_t *set, int signum);
```



## POSIX.

```
1. // POSIX
2. struct sigaction {
3.     void (*sa_handler)(int);
4.     void (*sa_sigaction)(int, siginfo_t *, void *);
5.     sigset_t sa_mask;
6.     int sa_flags; // SA_NODEFER, SA_RESETHAND, SA_SIGINFO,
SA_RESTART
7.     void (*sa_restorer)(void);
8. }

9. int sigemptyset(sigset_t *set);
10. int sigfillset(sigset_t *set);
11. int sigaddset(sigset_t *set, int signum);
12. int sigdelset(sigset_t *set, int signum);
13. int sigismember(const sigset_t *set, int signum);
```



## POSIX.

```
1. // POSIX
2. struct sigaction {
3.     void (*sa_handler)(int);
4.     void (*sa_sigaction)(int, siginfo_t *, void *);
5.     sigset_t sa_mask;
6.     int sa_flags; // SA_NODEFER, SA_RESETHAND, SA_SIGINFO,
SA_RESTART
7.     void (*sa_restorer)(void);
8. }

9. int sigemptyset(sigset_t *set);
10. int sigfillset(sigset_t *set);
11. int sigaddset(sigset_t *set, int signum);
12. int sigdelset(sigset_t *set, int signum);
13. int sigismember(const sigset_t *set, int signum);
14. int sigaction(int signum, const struct sigaction *act, struct
sigaction *oldact);
```



## siginfo\_t.

```
1.  union sigval {
2.      int sival_int;
3.      void *sival_ptr;
4.  };
5.  typedef struct {
6.      int si_signo;
7.      int si_code; /* SI_USER, SI_KERNEL, SI_QUEUE, SI_TIMER */
8.      union sigval si_value;
9.      int si_errno;
10.     pid_t si_pid;
11.     uid_t si_uid;
12.     void *si_addr;
13.     int si_status;
14.     int si_band;
15. } siginfo_t;
```



## Отправка и ожидание сигнала.

```
1.  int kill(pid_t pid, int signum);
2.  int sigqueue(pid_t pid, int sig, const union sigval value);

3.  int raise(int signum); // Синхронно.
4.  int abort(); // SIGABRT
5.  unsigned alarm(unsigned secs); // SIGALRM

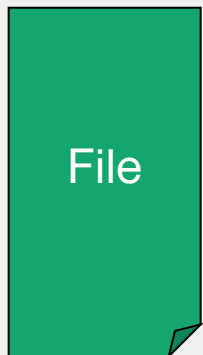
6.  int pause();
7.  int sigwait(const sigset_t *set, int *signum);
```



## Безопасность.

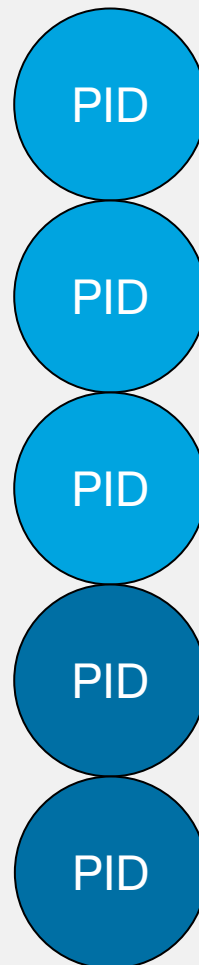
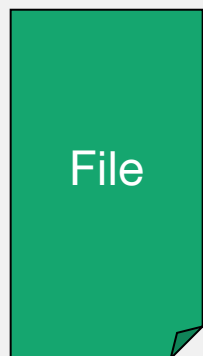
```
1. volatile sig_atomic_t GlobalVariable;
```

# Блокировка

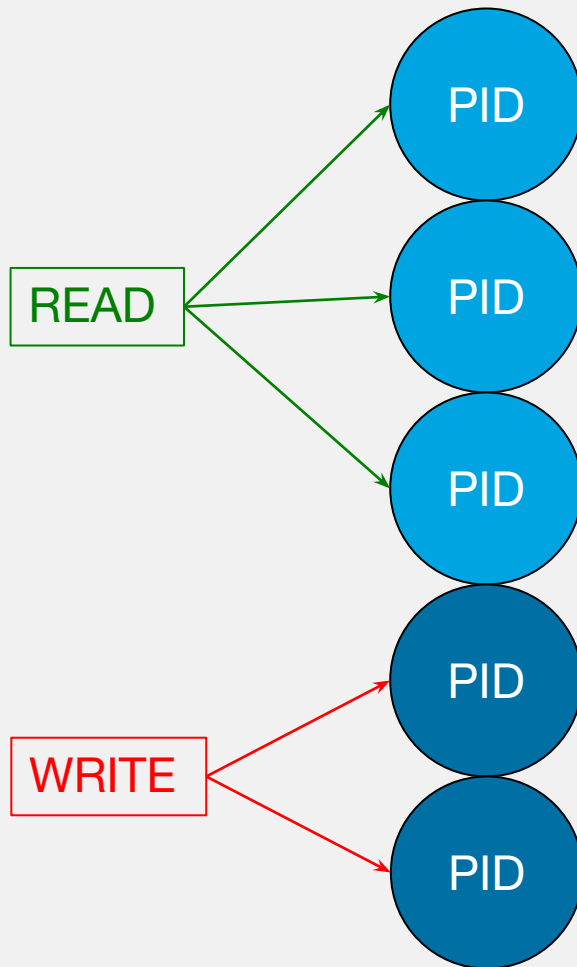
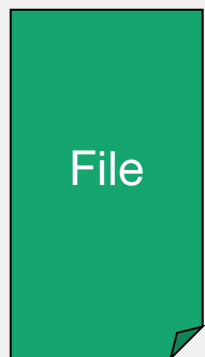




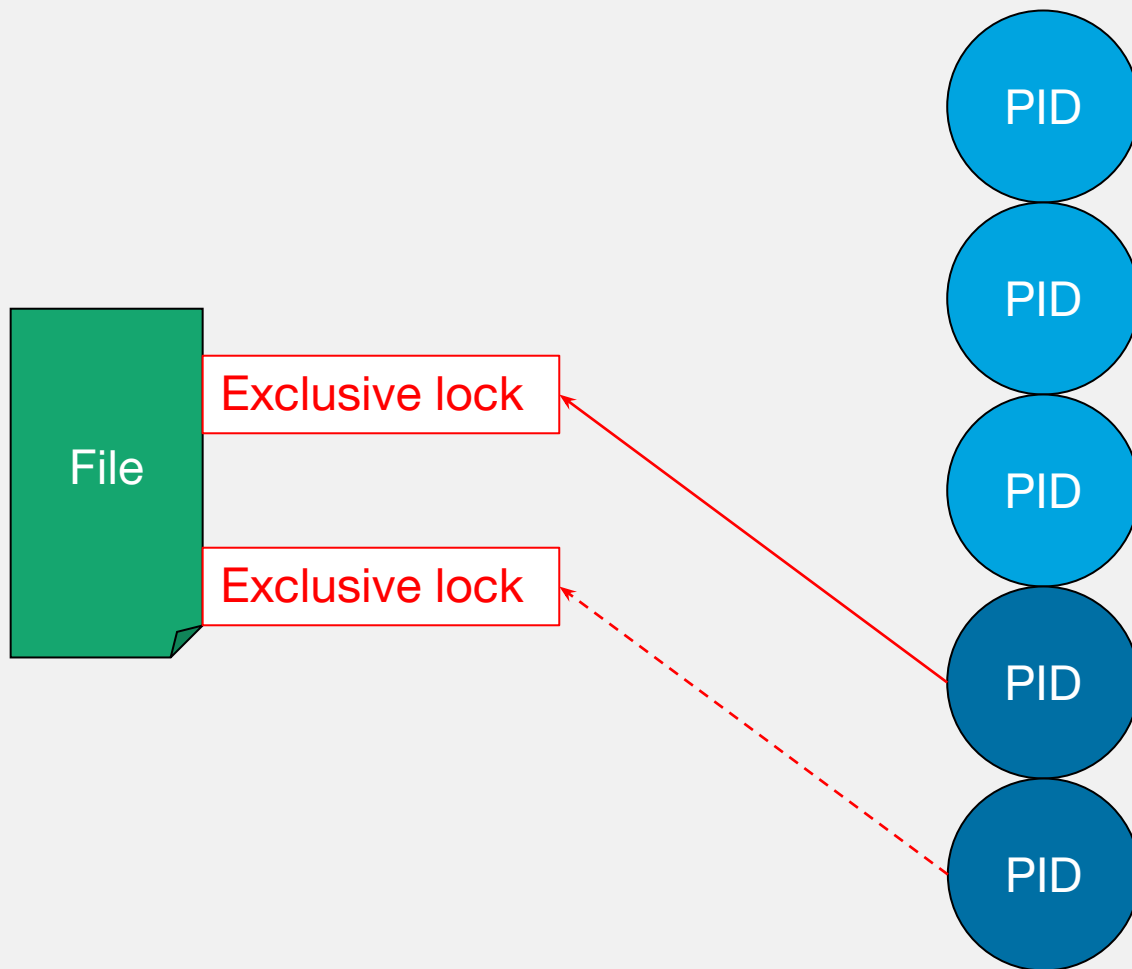
# Блокировка



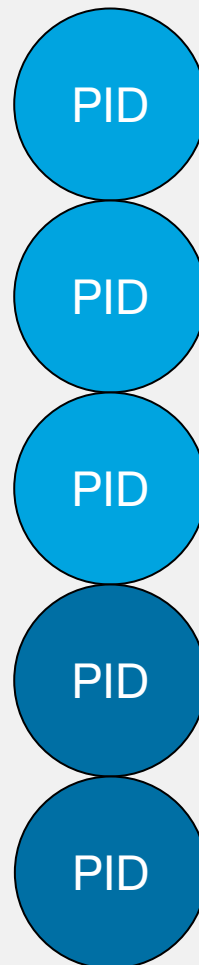
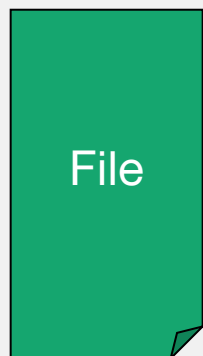
# Блокировка



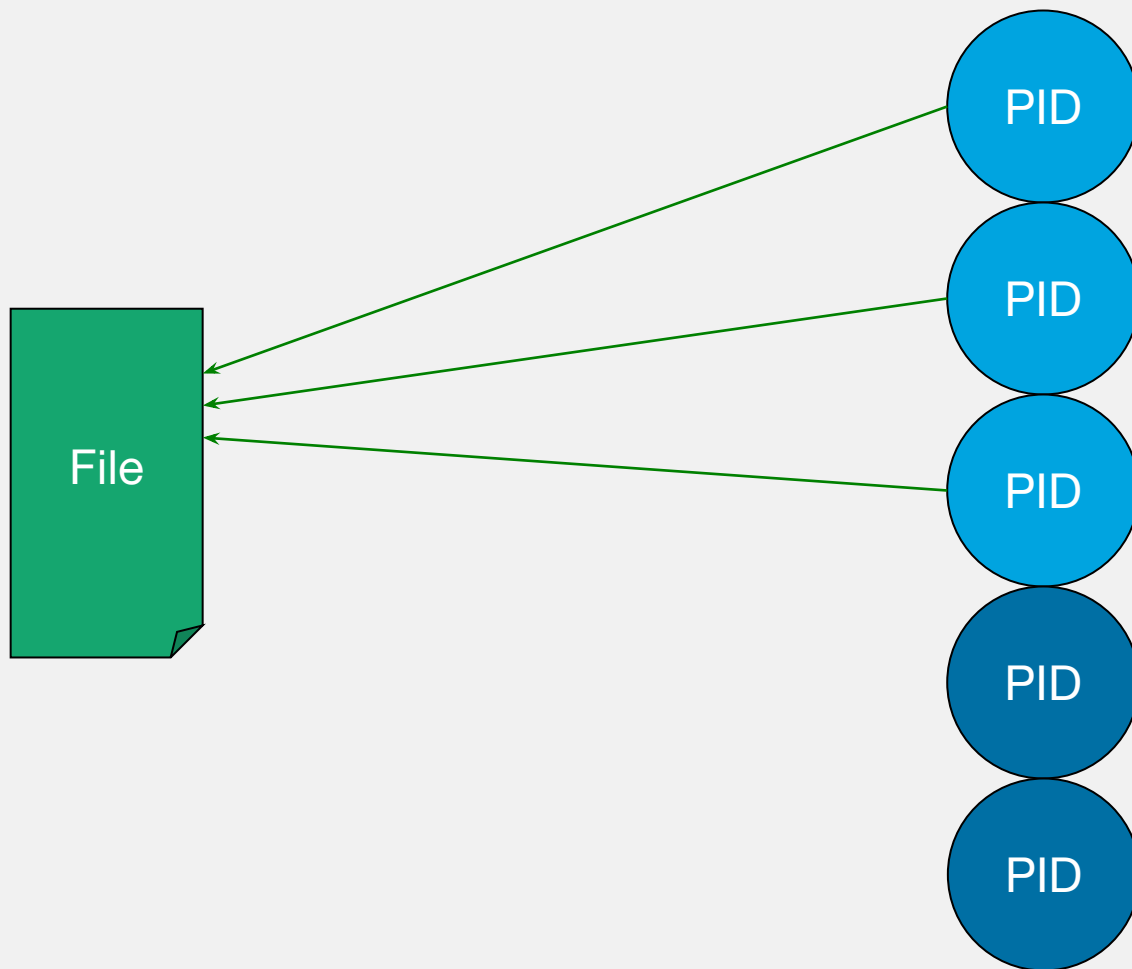
# Блокировка



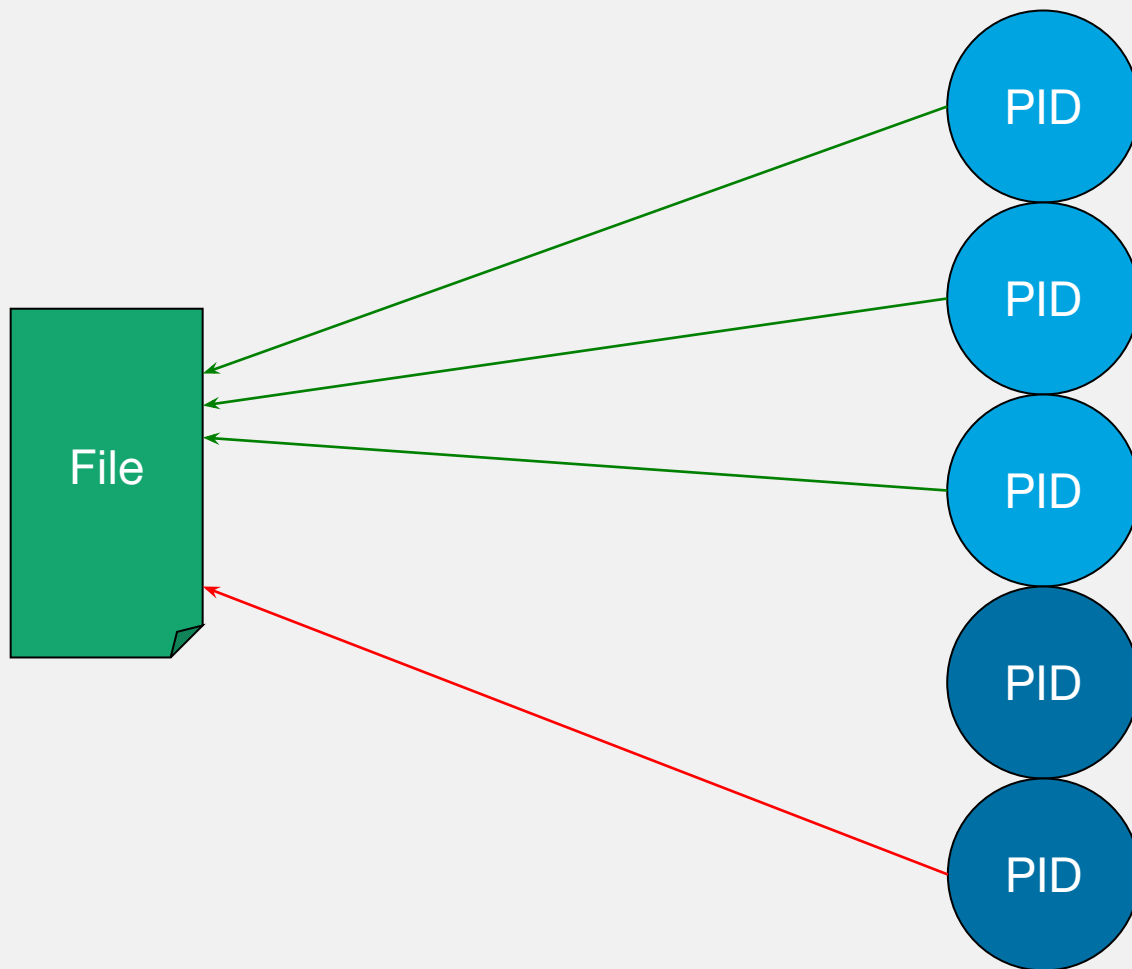
# Блокировка



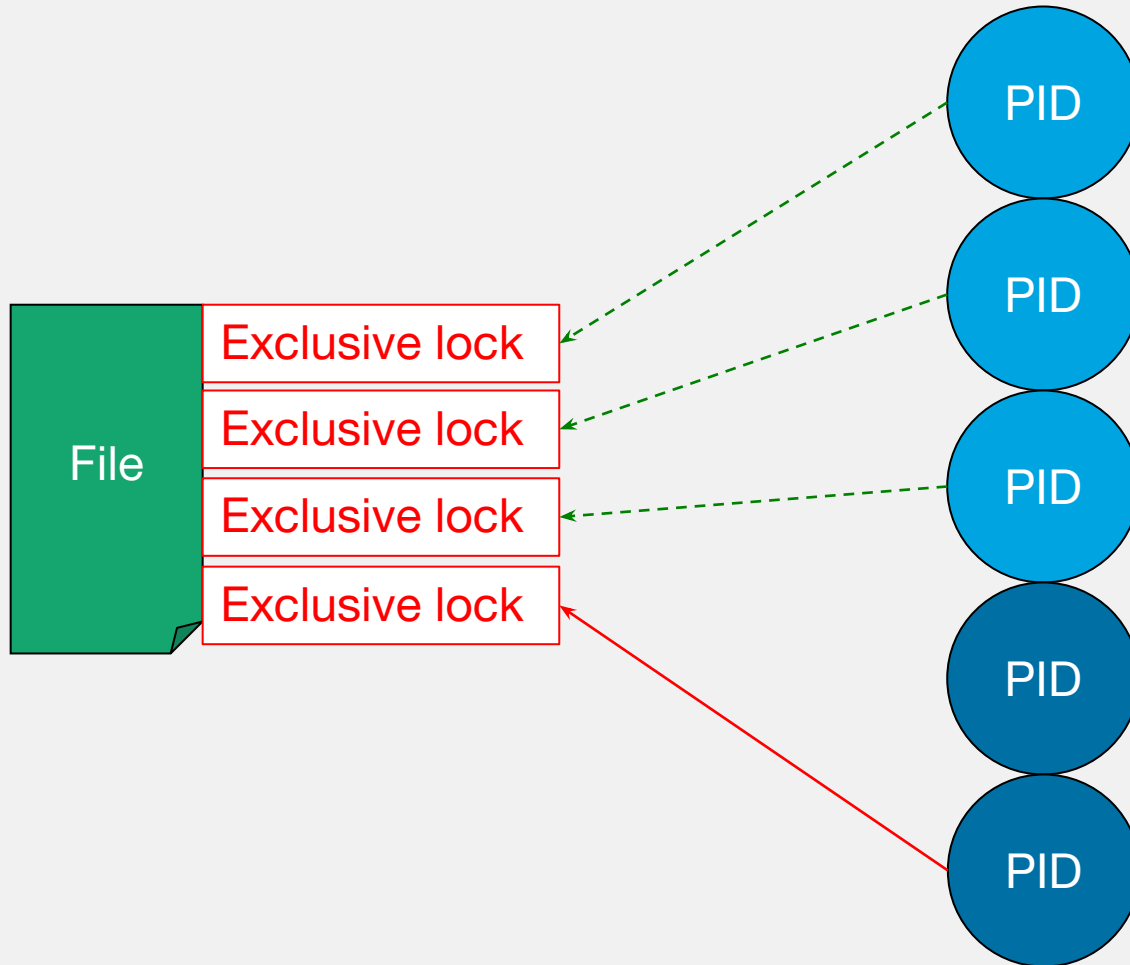
# Блокировка



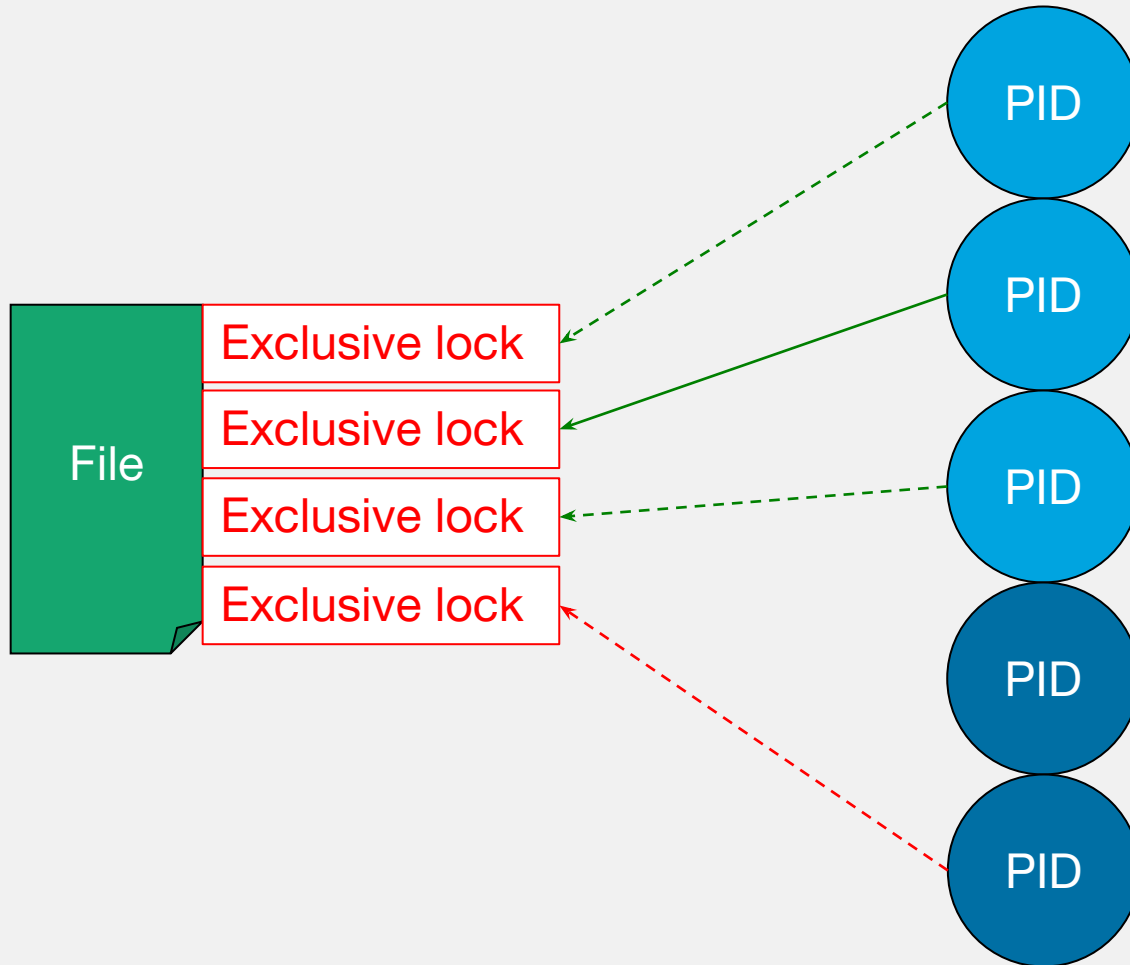
# Блокировка



# Блокировка

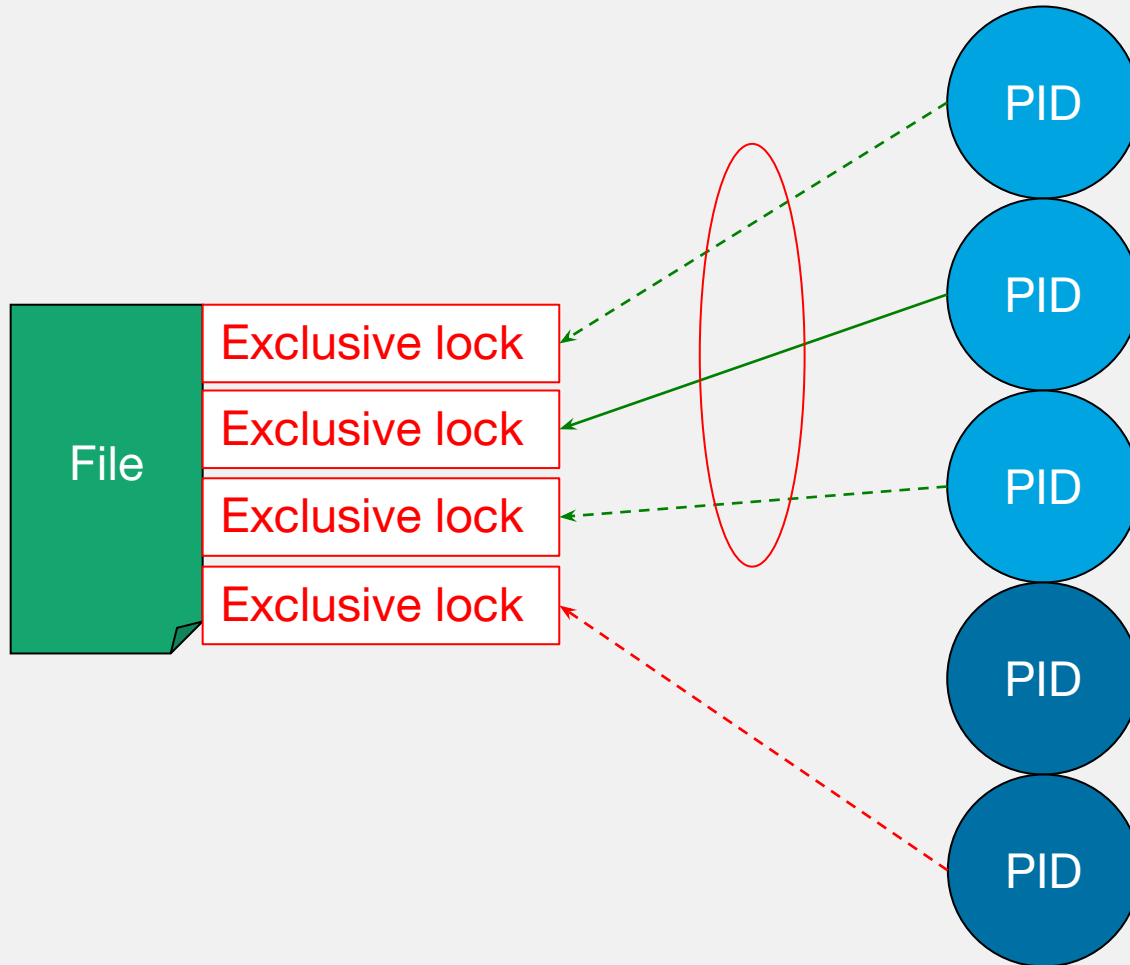


# Блокировка

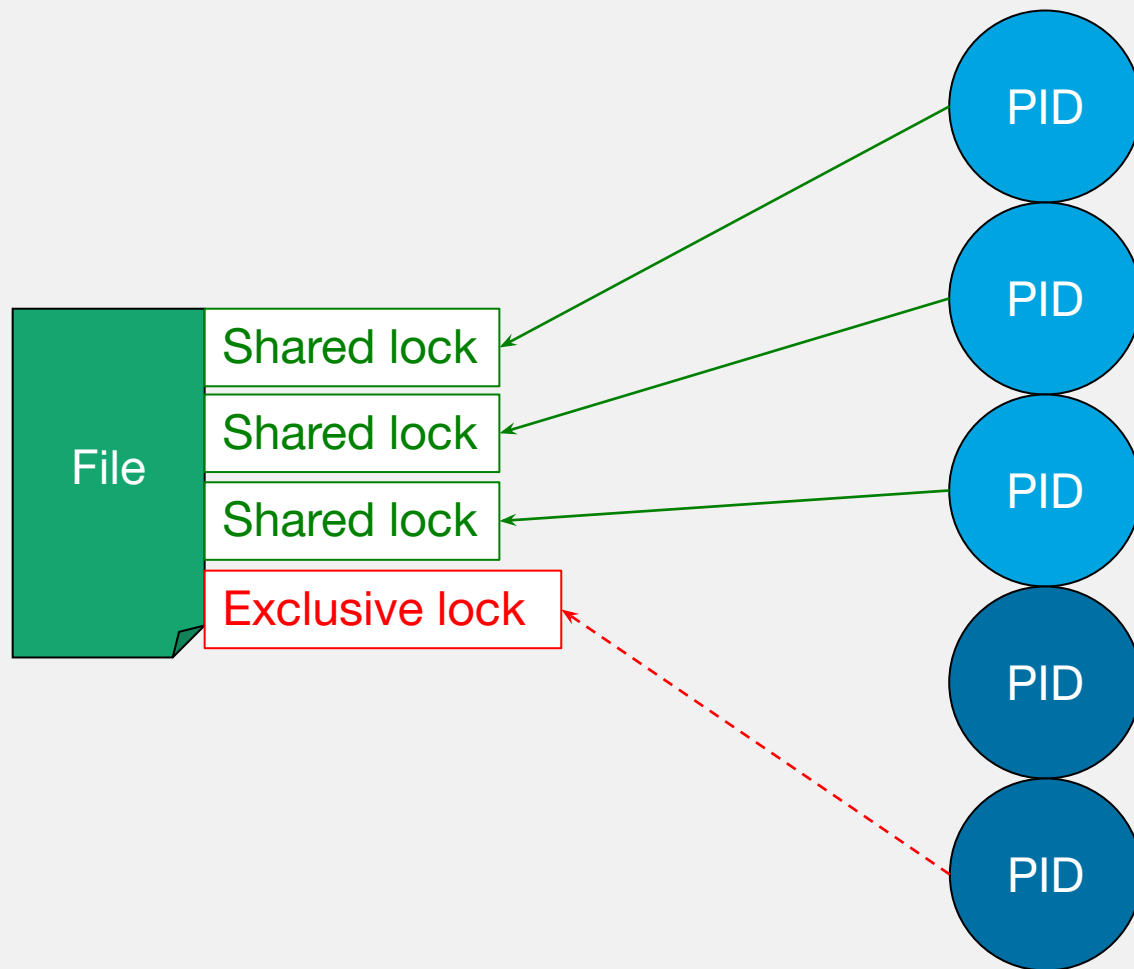




# Блокировка



# Блокировка





## Блокировки в коде.

1. `fcntl(fd, LOCK_EX);`
2. `fcntl(fd, LOCK_SH);`
3. `fcntl(fd, LOCK_UN);`
  
4. `fcntl(fd, LOCK_EX | LOCK_NB);`
5. `fcntl(fd, LOCK_SH | LOCK_NB);`



# Файлы блокировок



## Что мы хотим?

```
1.  if(lock("filename"))
2.  {
3.      // Какие-то действия
4.      unlock("filename");
5.  }
6.  else
7.  {
8.      // ...
9.  }
```



## Реализация lock().

```
1.  bool lock(char *filename)
2.  {
3.      int fd;
4.      if(fd = open(filename, O_WRONLY | O_CREATE | O_EXCL, 0))
5.          == -1)
6.          {
7.              return false;
8.          }
9.          close(fd);
10.         return true;
11.     }
```



## Реализация lock().

```
1.  bool lock(char *filename)
2.  {
3.      int fd;
4.      if(fd = open(filename, O_WRONLY | O_CREATE | O_EXCL, 0))
5.          == -1)
6.          {
7.              return false;
8.          }
9.          close(fd);
10.         return true;
11.     }
```



## Реализация lock().

```
1.  bool lock(char *filename)
2.  {
3.      int fd;
4.      if(fd = open(filename, O_WRONLY | O_CREATE | O_EXCL, 0))
5.          == -1)
6.          {
7.              return false;
8.          }
9.          close(fd);
10.         return true;
11.     }
```



## Реализация `unlock()`.

```
1.  bool unlock(char *filename)
2.  {
3.      unlink(filename);
4.      return true;
5.  }
```



# Общие смещения в файлах

---



Какой-то файл...

# Общие смещения в файлах



Процесс 1

Какой-то файл...

# Общие смещения в файлах



Процесс 1

Какой-то файл...

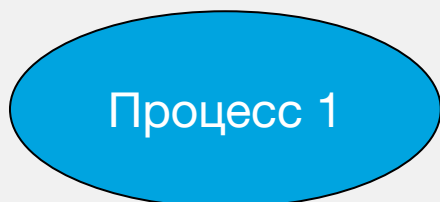
# Общие смещения в файлах



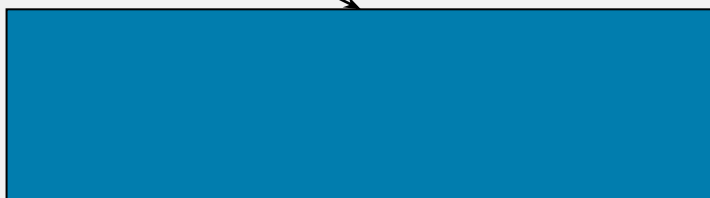
Процесс 1

Какой-то файл...

# Общие смещения в файлах

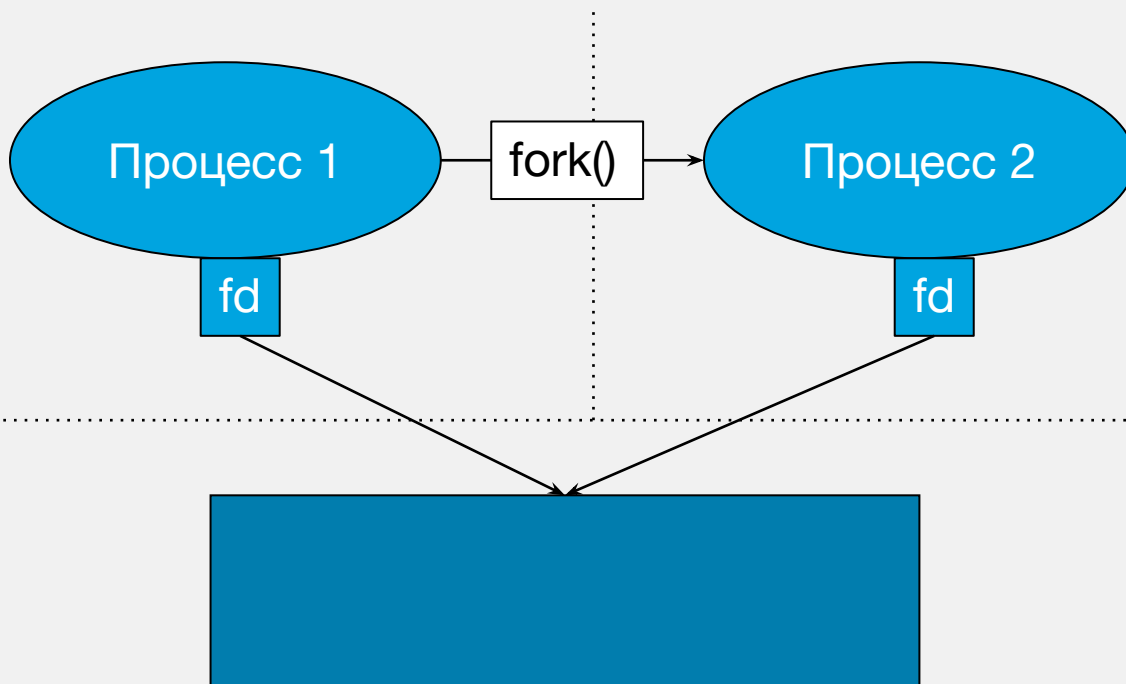


fd



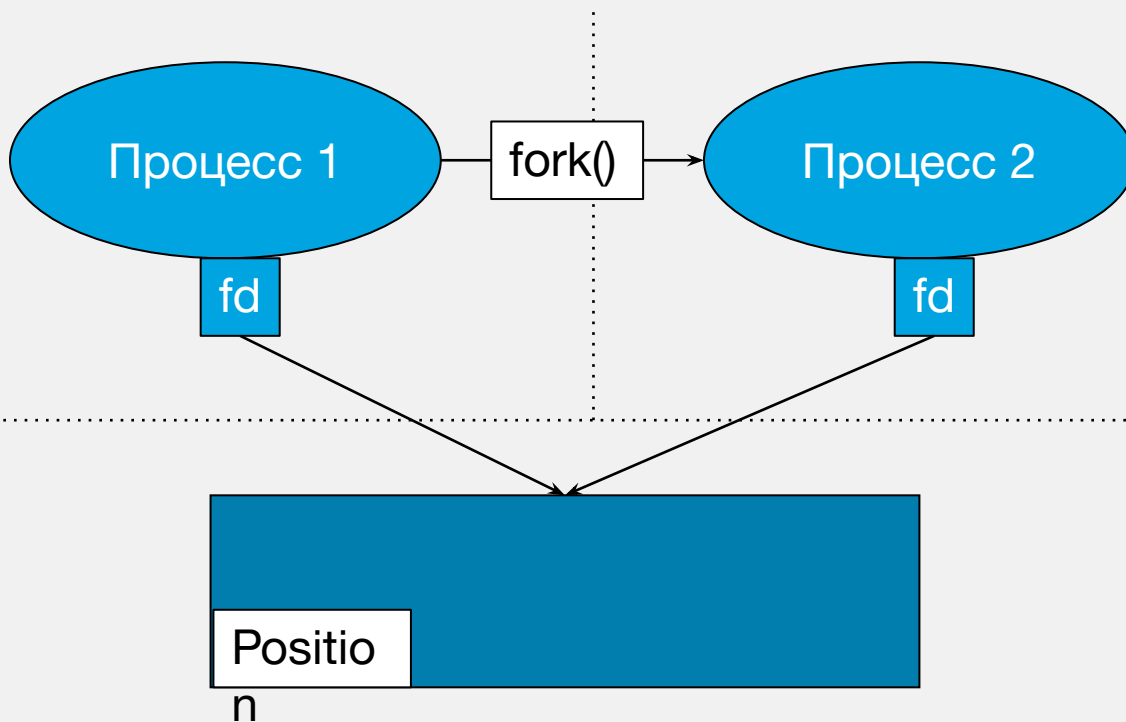
Какой-то файл...

# Общие смещения в файлах



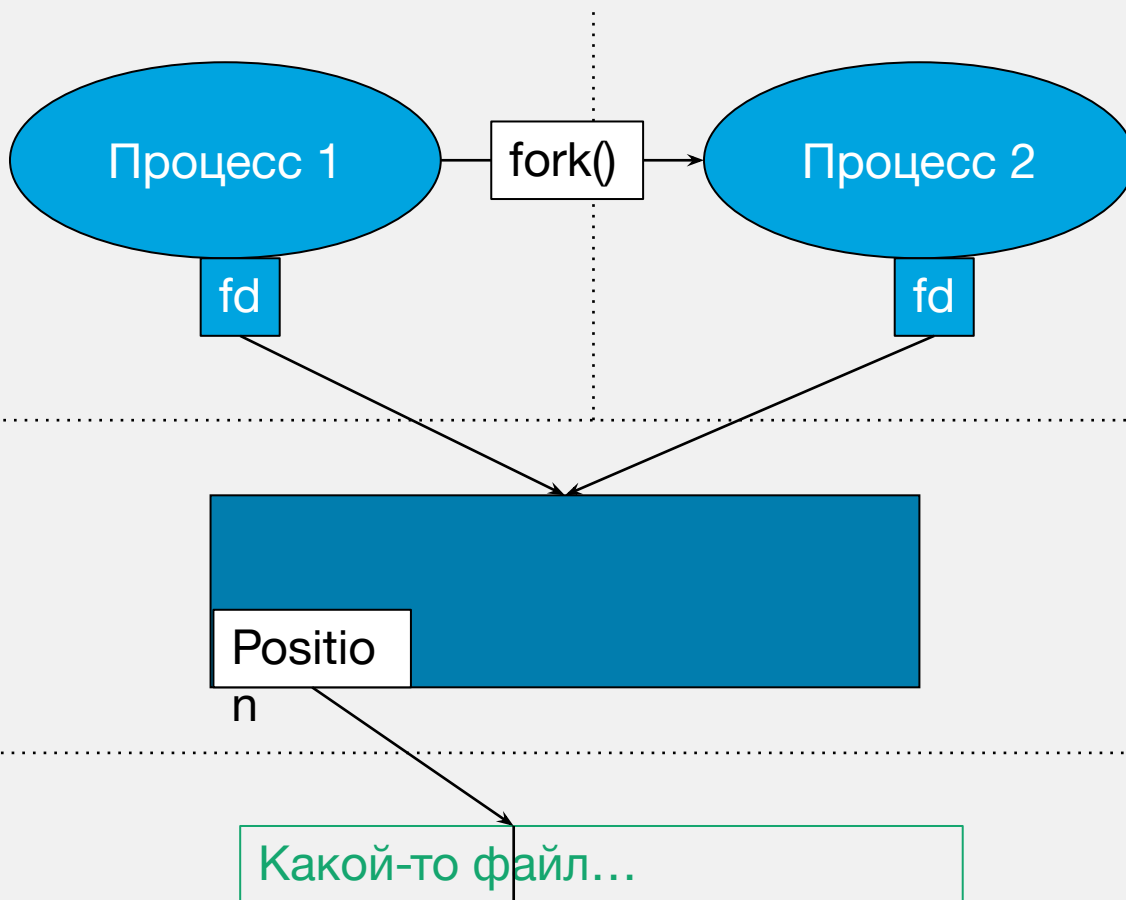
Какой-то файл...

# Общие смещения в файлах



Какой-то файл...

# Общие смещения в файлах

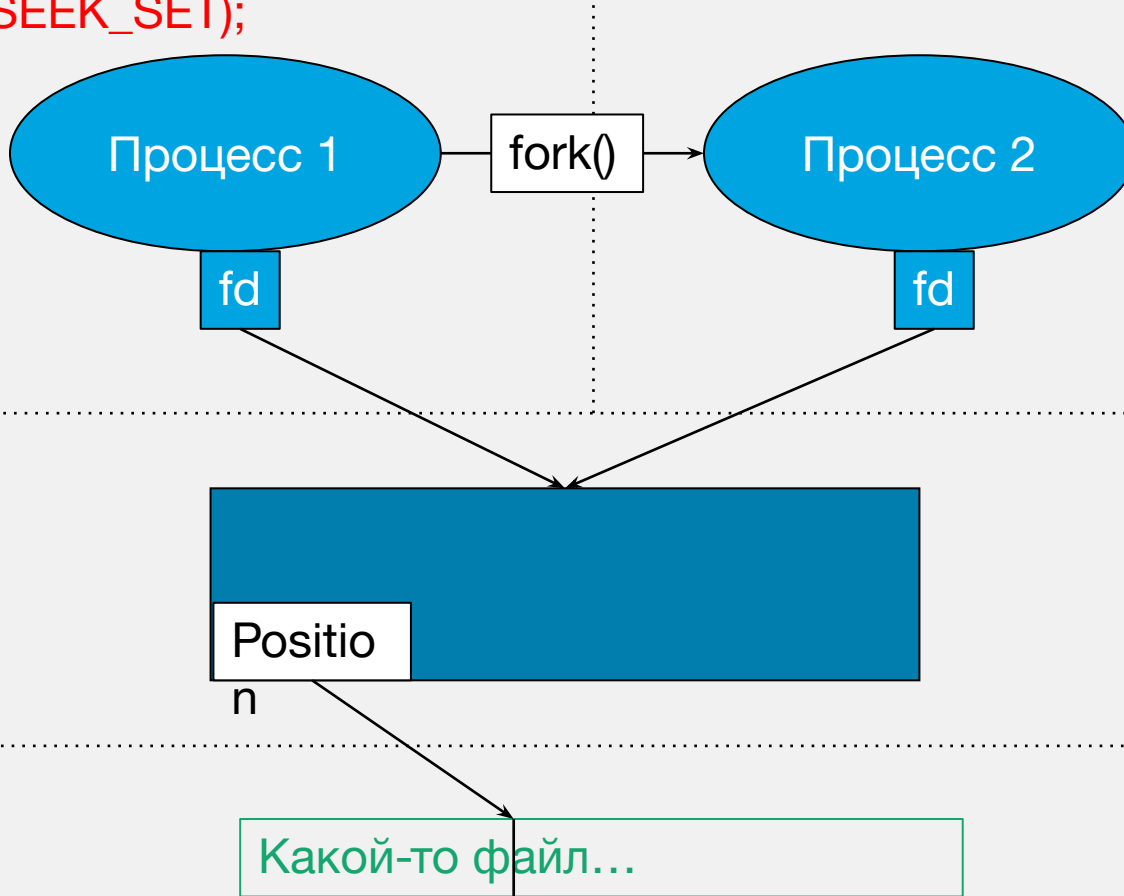




# Общие смещения в файлах



`lseek(fd, NUMBER,  
SEEK_SET);`

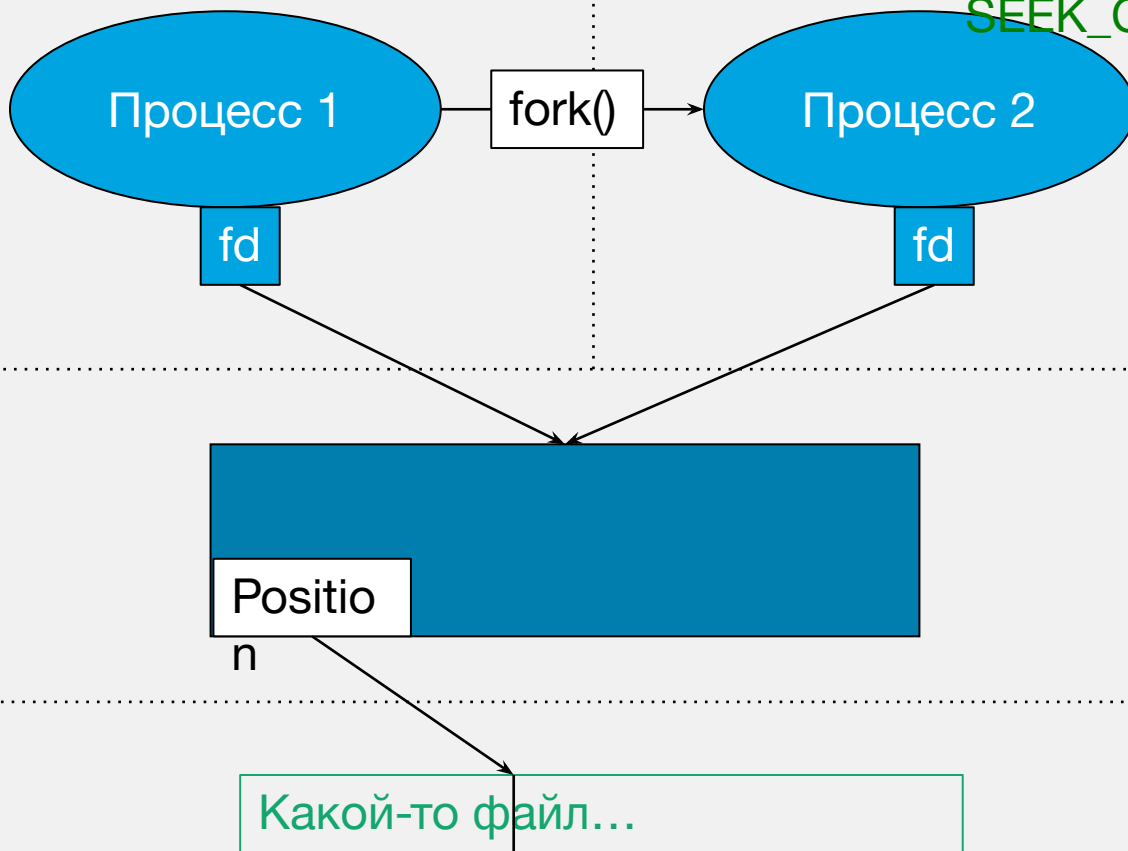


# Общие смещения в файлах



`lseek(fd, NUMBER,  
SEEK_SET);`

`lseek(fd, 0,  
SEEK_CUR);`



# Неименованные каналы



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

# Неименованные каналы



fd[1]	fd[0]
-------	-------

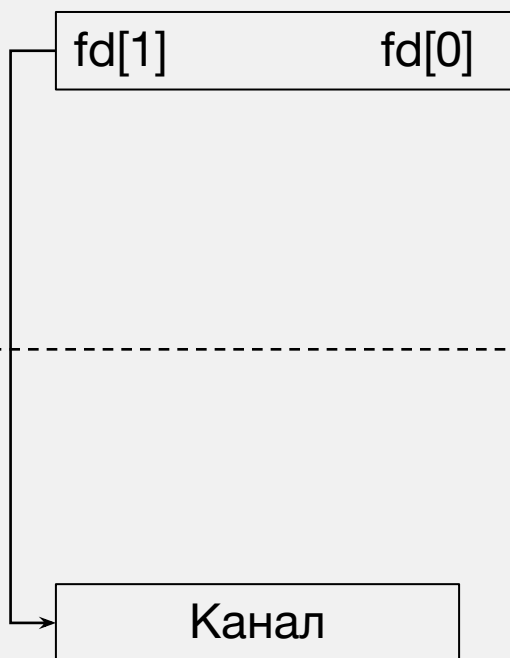
# Неименованные каналы



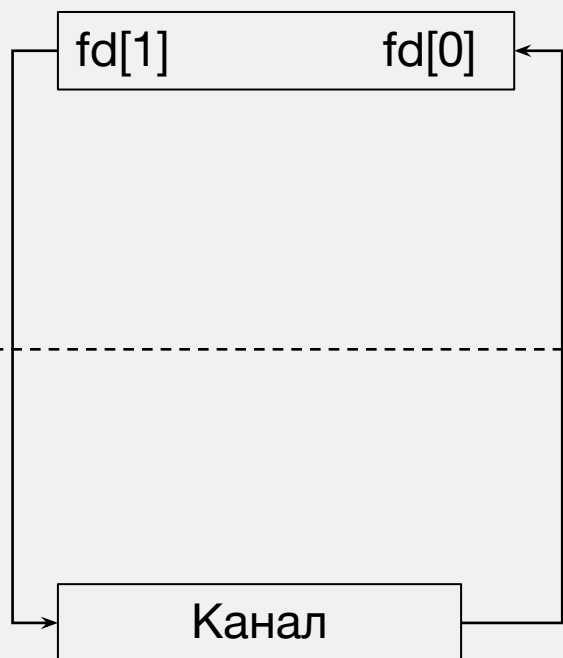
fd[1]      fd[0]

Канал

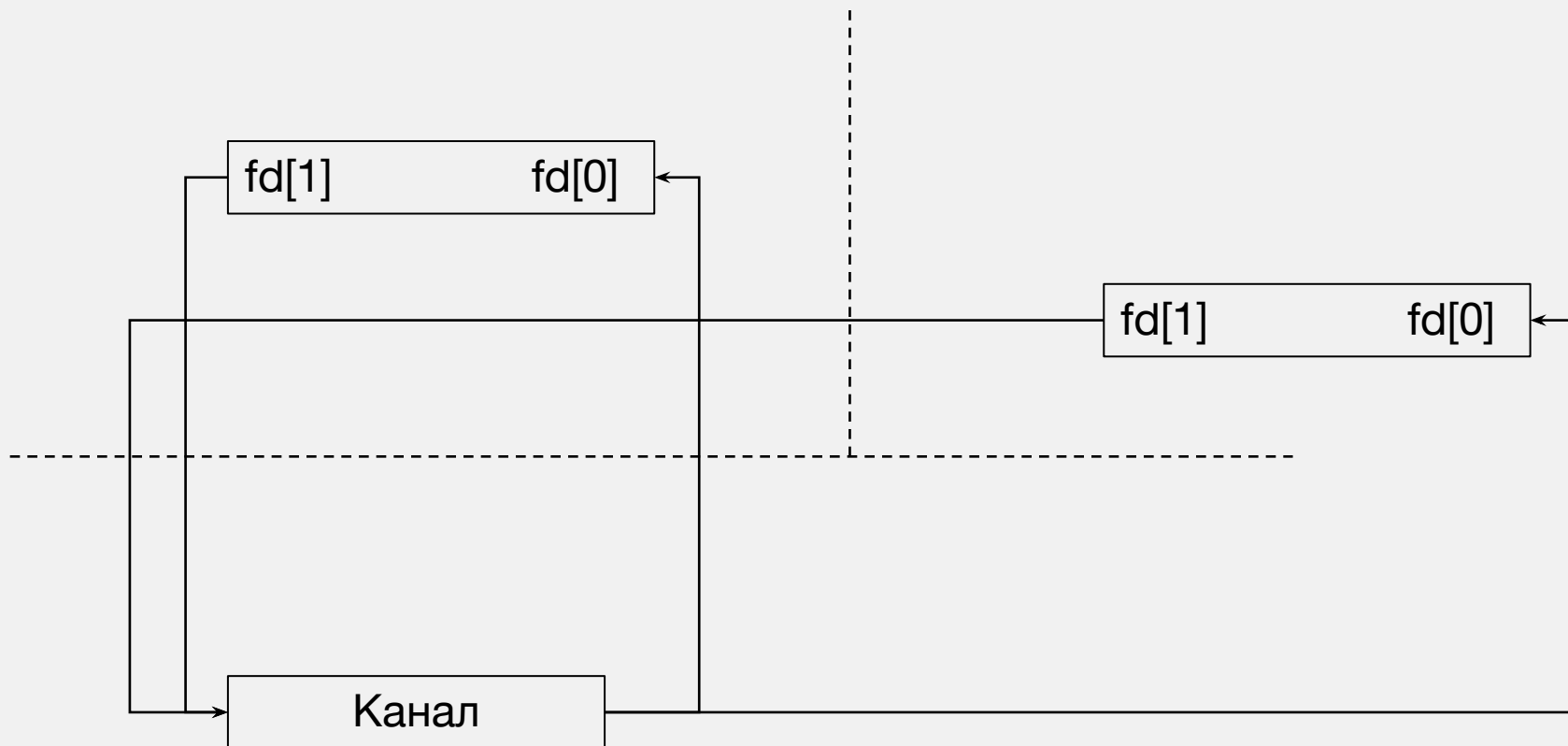
# Неименованные каналы



# Неименованные каналы

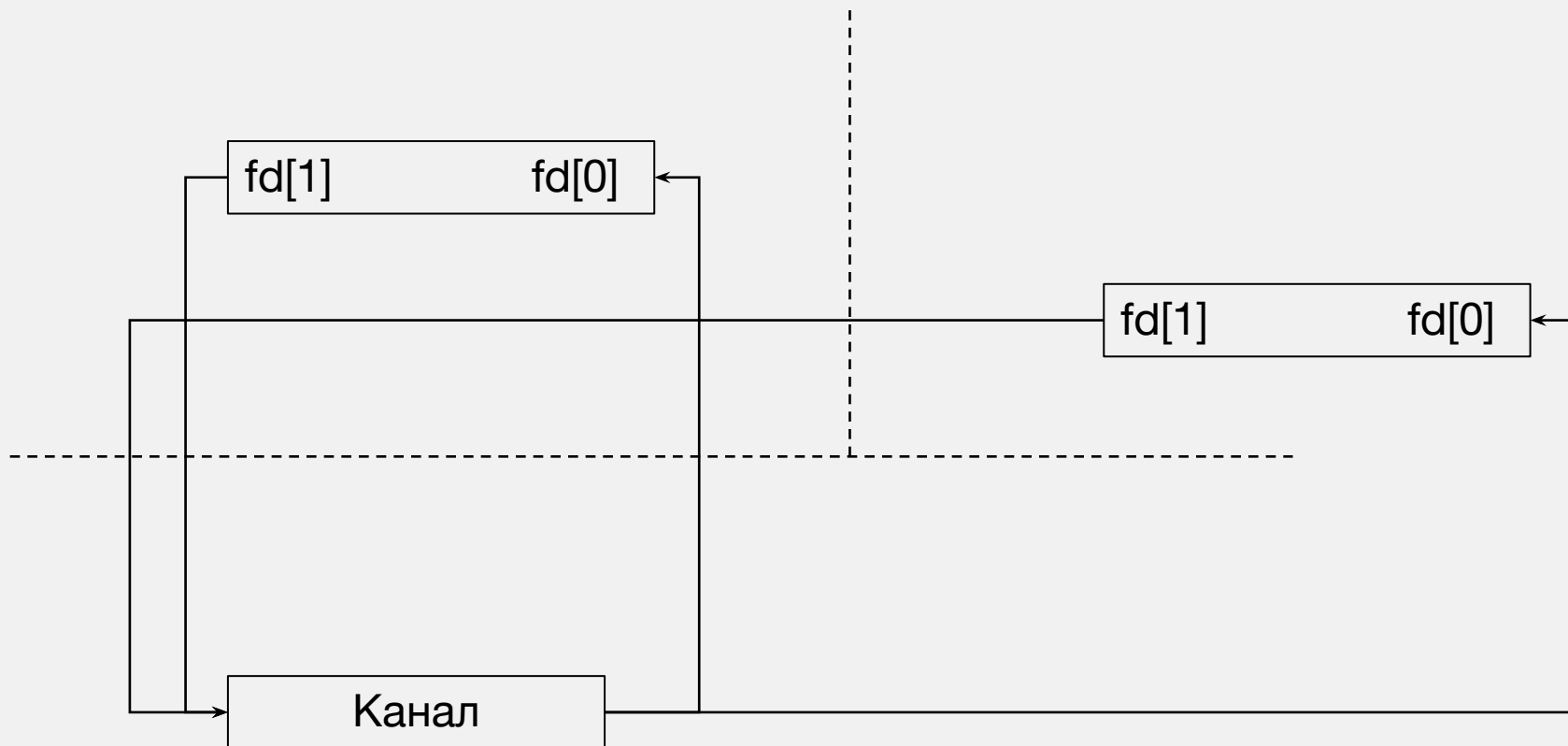


# Неименованные каналы

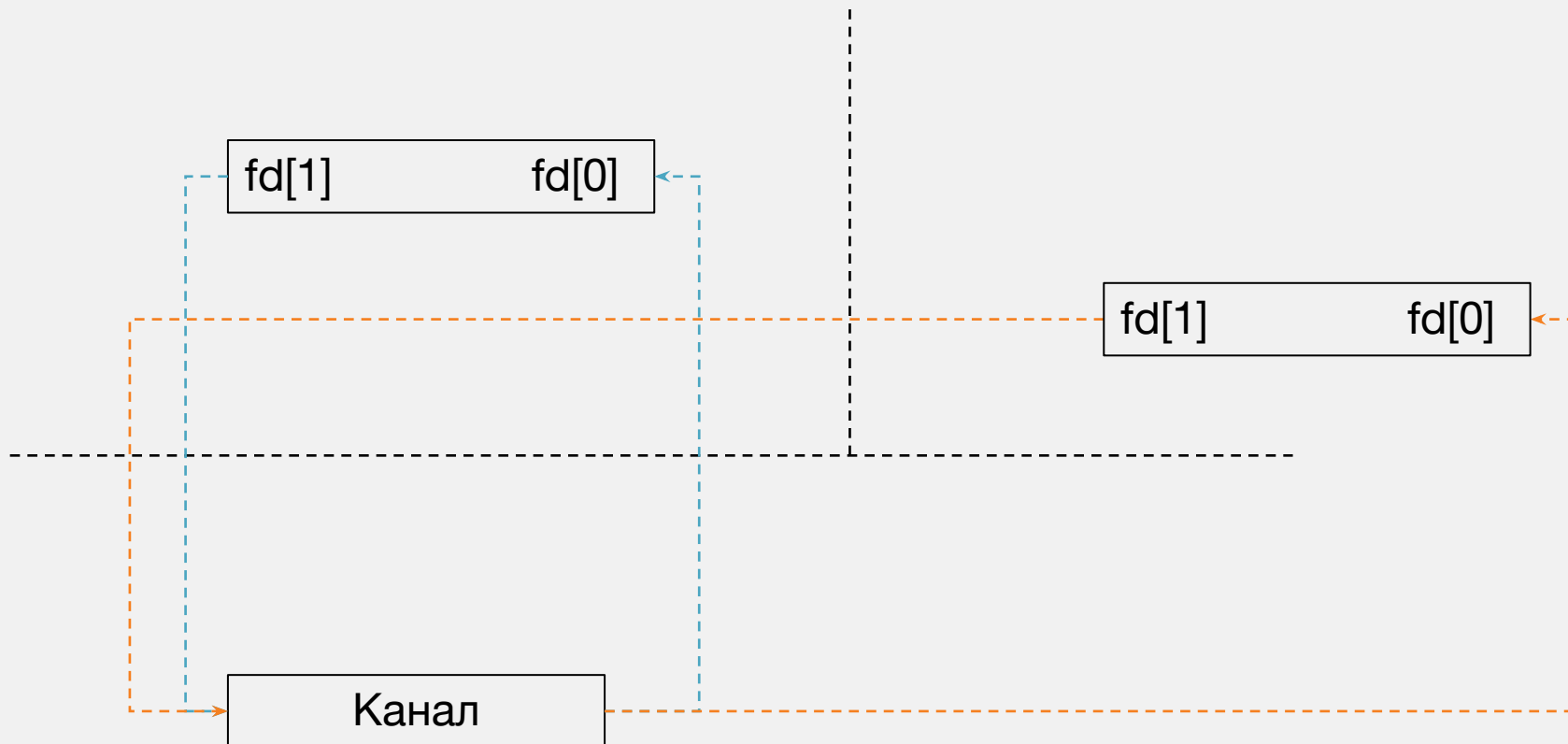




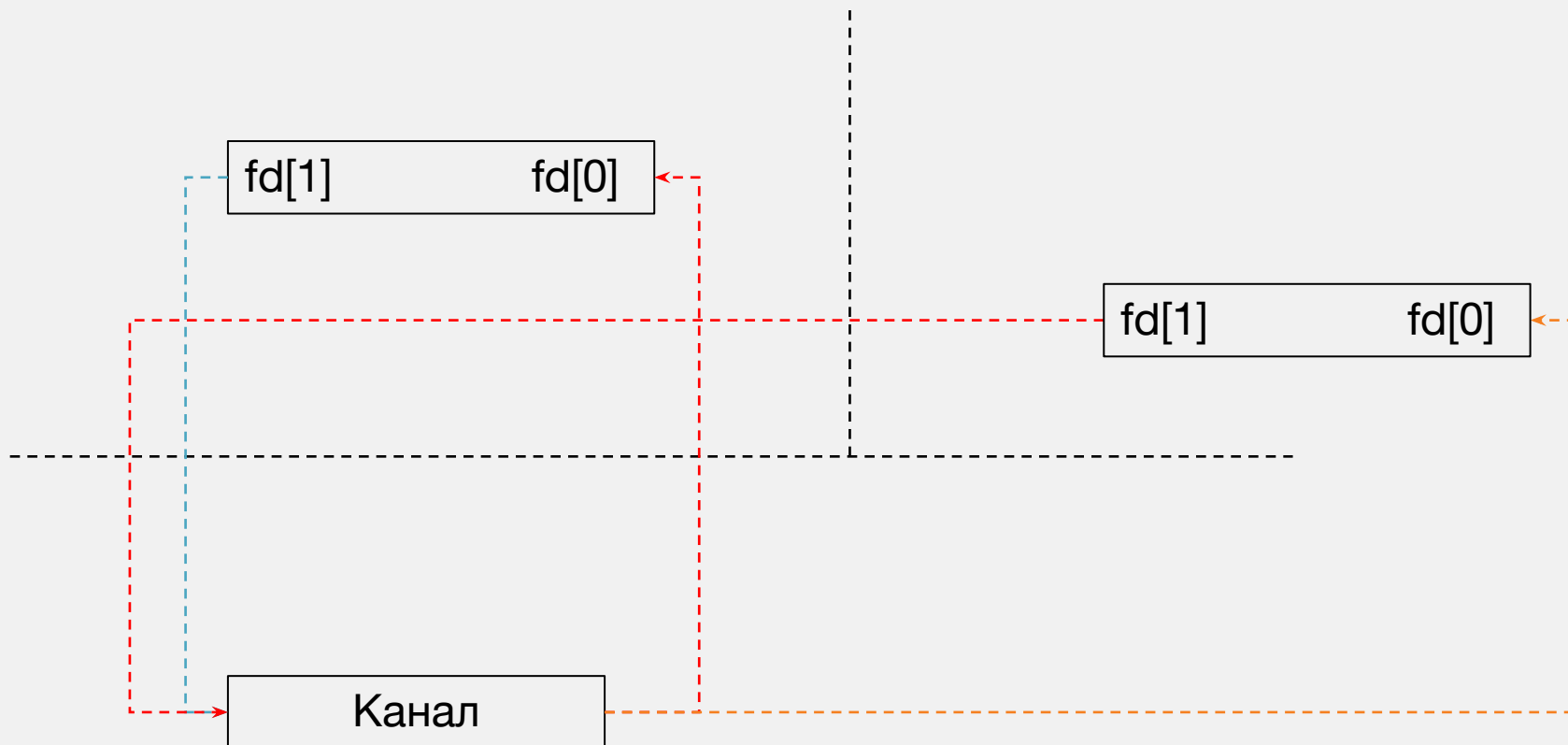
# Неименованные каналы



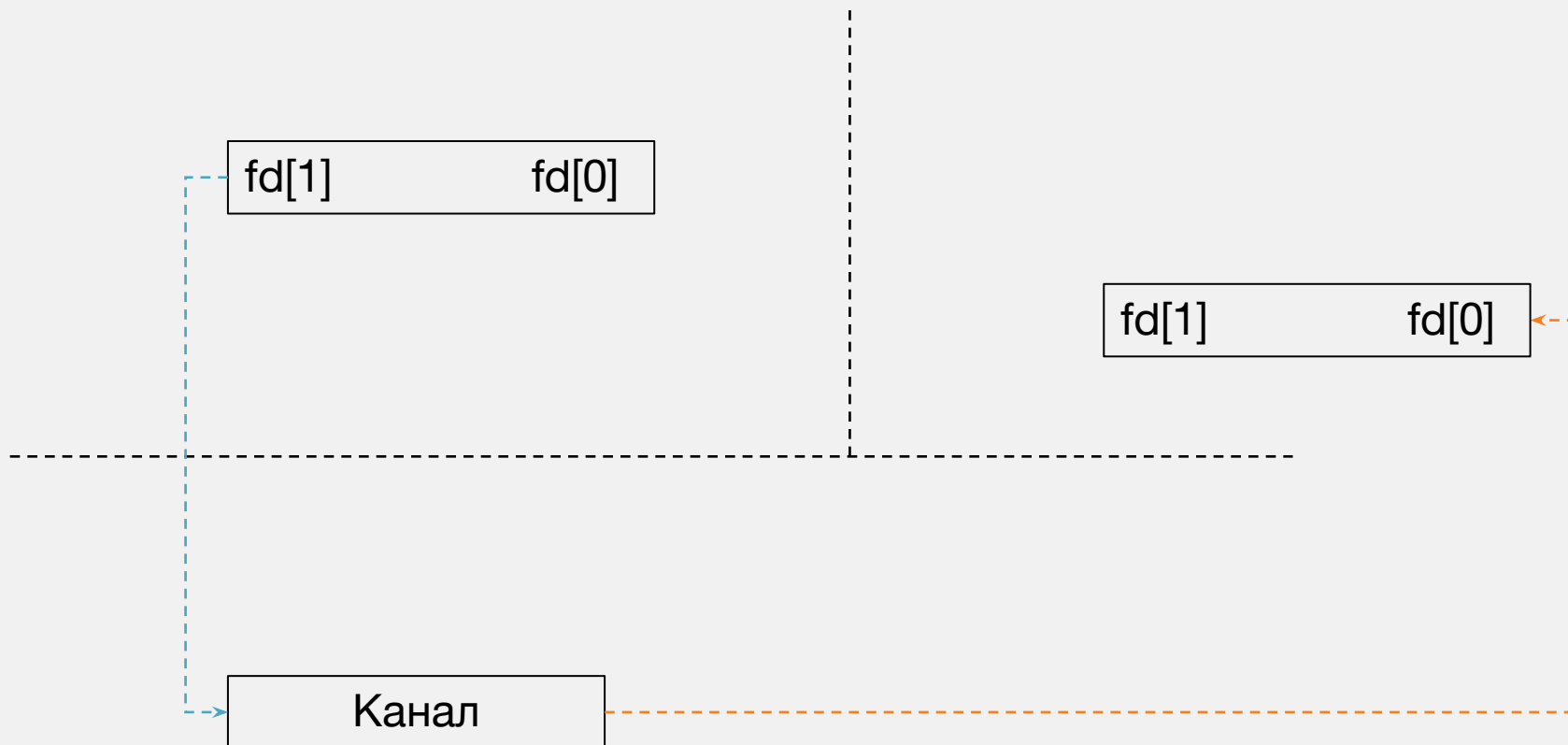
# Неименованные каналы



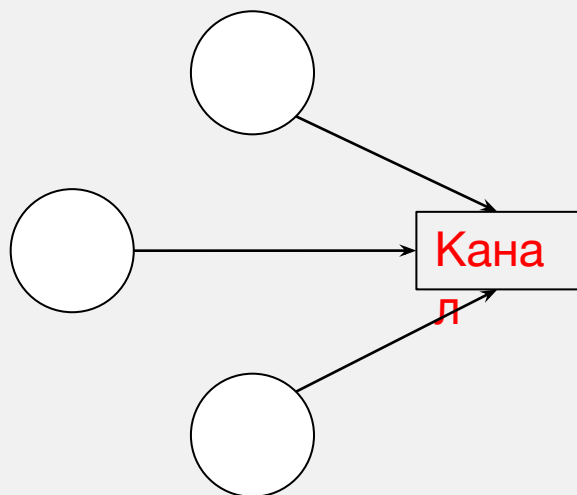
# Неименованные каналы



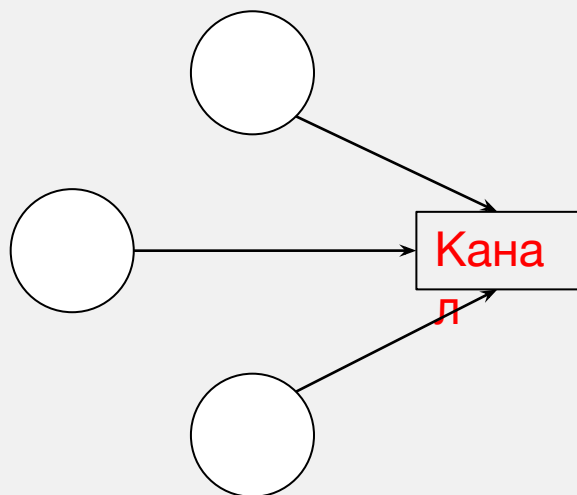
# Неименованные каналы



# Неименованные каналы

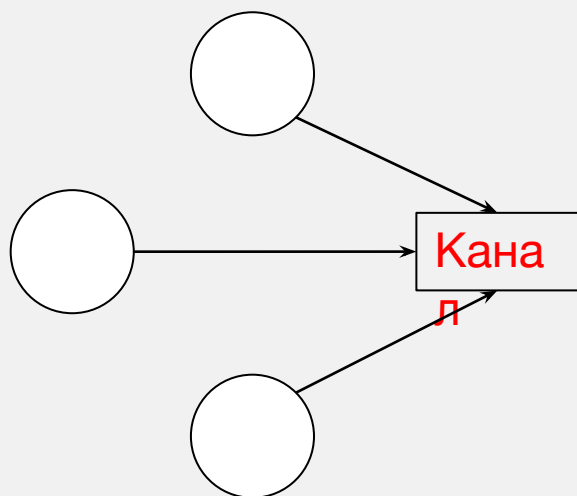


# Неименованные каналы



PIPE\_B  
UF

# Неименованные каналы



PIPE\_B  
UF

```
long v = fpathconf(pfd[0],  
_PC_PIPE_BUF);
```



# Неименованные каналы



## Пример конвейера.

```
1.  who | sort | uniq | wc
```



# Неименованные каналы

---



**who**

stdout

stdin **sort**

stdout

stdin **uniq** stdout

stdin

**wc**

# Неименованные каналы



**who**  
stdout

stdin **sort**  
stdout

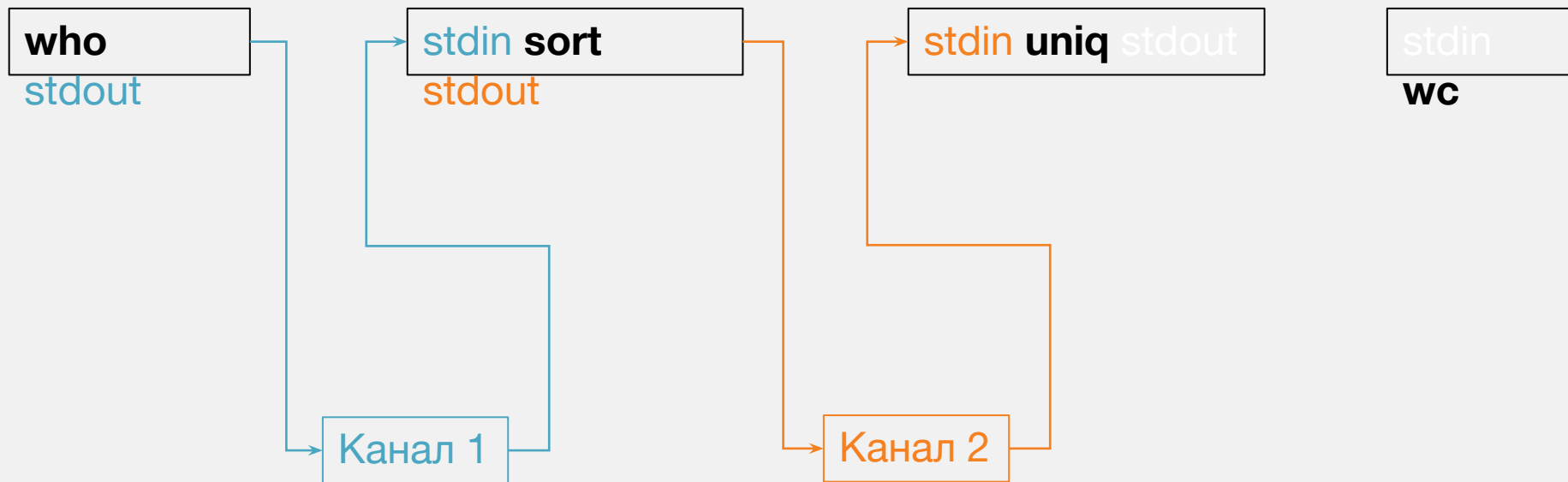
stdin **uniq** stdout

stdin  
**wc**

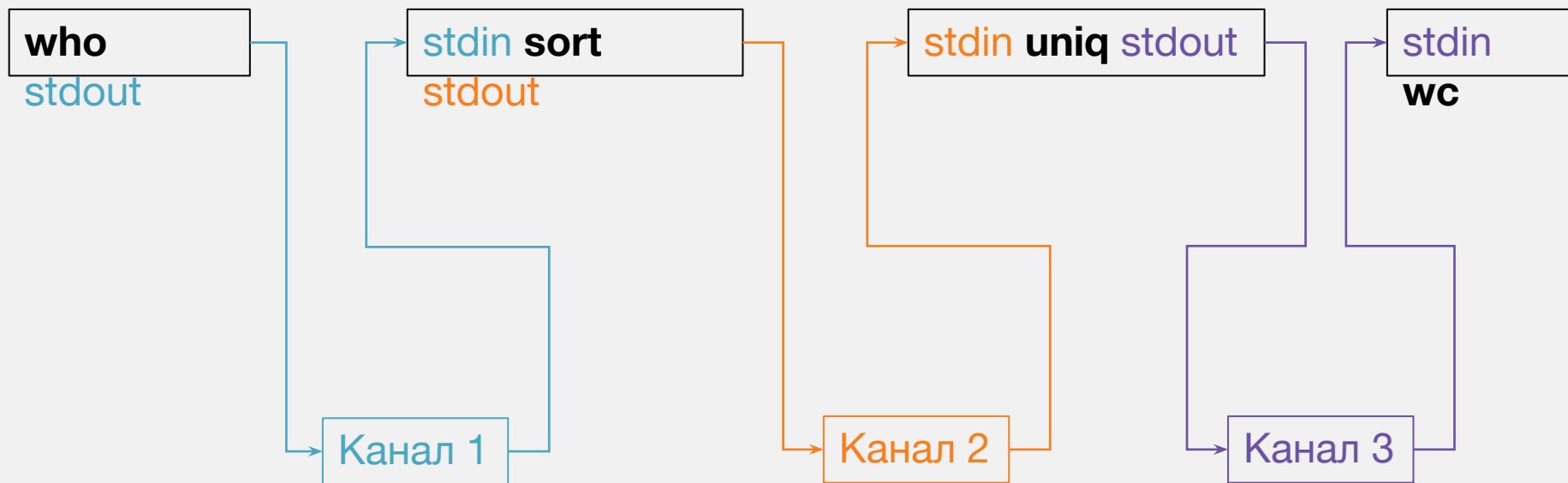
# Неименованные каналы



# Неименованные каналы



# Неименованные каналы





## Дублирование дескрипторов.

1. `int dup(int oldfd);`
2. `int dup2(int oldfd, int newfd); // dup2(N, N) = N`
3. `int dup3(int oldfd, int newfd, int flags /* O_CLOEXEC */);`



## Пример конвеера.

```
1. void who_wc() {
2.     int pfd[2];
3.     pipe(pfd);
4.     if(!fork()) {
5.         close(STDOUT_FILENO);
6.         dup2(pfd[1], STDOUT_FILENO);
7.         close(pfd[0]); close(pfd[1]);
8.         execlp("who", "who", NULL); }
9.     if(!fork()) {
10.        close(STDIN_FILENO);
11.        dup2(pfd[0], STDIN_FILENO);
12.        close(pfd[0]); close(pfd[1]);
13.        execlp("wc", "wc", "-l", NULL); }
14.    close(pfd[0]); close(pfd[1]);
15.    wait(NULL); wait(NULL);
16. }
```



# Неименованные каналы



**popen и pclose.**

1. `FILE *popen(const char *command, const char *type);`
2. `int pclose(FILE *stream);`





# Неименованные каналы



## Двусторонние каналы.

```
1.  int socketpair(int d, /* AF_UNIX */  
2.     int type,  
3.     int protocol,  
4.     int sv[2]);
```



## FIFO.

```
1. int mkfifo(const char *path,  
2. mode_t perms);
```



**Спасибо за  
внимание!**

**Дмитрий Калугин-Балашов**

[rvncerr@rvncerr.org](mailto:rvncerr@rvncerr.org)