

# Практикум на ЭВМ

## Системные вызовы.

### Ввод, вывод, fork, exec.

Баев А.Ж.

Казахстанский филиал МГУ

18 сентября 2020

# План

- 1 Справка
- 2 Системные вызовы
- 3 read, write
- 4 Дескрипторы
- 5 Директории
- 6 Процессы
- 7 Дочерний процесс
- 8 Замена процесса
- 9 Ассемблер

## Справочные страницы man

- ❶ Исполняемые программы или команды оболочки (shell)
- ❷ Системные вызовы (функции, предоставляемые ядром)
- ❸ Библиотечные вызовы (функции, предоставляемые программными библиотеками)
- ❹ Специальные файлы (обычно находящиеся в каталоге /dev)
- ❺ Форматы файлов и соглашения, например о /etc/passwd
- ❻ Игры
- ❼ Разное (включает пакеты макросов и соглашения), например man(7), groff(7)
- ❽ Команды администрирования системы (обычно, запускаемые только суперпользователем)
- ❾ Процедуры ядра [нестандартный раздел]

## Разделы

Можно посмотреть информацию о команде shell, системном вызове и прочее.

```
1 man cd
2 man fork
3 man man
```

Можно делать приближенный поиск и уточнять раздел поиска

```
1 man -k read
2 man -k . -s 2
```

Выход из справки: q.

Нам нужен второй раздел

```
1 man 2 mkdir
2 man mkdir
```

## Немного о типах

Используйте те типы, которые указаны в описании функции.  
Для указания количества или размера используются

```
1 size_t  
2 ssize_t
```

которые являются аналогами *unsigned long* и *long* соответственно.  
Отрицательные значения, как правило, должны сигнализировать об ошибке.

## Обработка ошибок

В случае ошибки (возвращает отрицательно число), номер ошибки сохраняется в `errno`.

```
1 #include <stdio.h>
2
3 void perror(const char *s);
```

Выводим текст ошибки со своими комментариями.

## Системный ввод и вывод

Низкоуровневый ввод и вывод:

```
1 #include <unistd.h>
2
3 ssize_t read(int fd, void *buf, size_t count);
4 ssize_t write(int fd, void *buf, size_t count);
```

здесь fd — файловый дескриптор (0 - ввод, 1 - вывод, 2 - ошибки, остальные - по умолчанию закрыты), buf — указатель на буфер при чтении или записи, count — максимальный размер.

## Пример

```
1  int main(int argc, char **argv) {
2      char buf[10];
3      size_t count = 10;
4      size_t len = read(0, buf, count);
5      if (len < 0) {
6          perror("My comment of failed read()");
7          return 1;
8      }
9      if (write(1, buf, len) < 0) {
10         perror("My comment of failed write()");
11         return 1;
12     }
13     return 0;
14 }
```

Если попытаться вывести в дескриптор 3. то будет

```
1  My comment of failed write(): Bad file descriptor
```



## Файл

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <fcntl.h>
4 #include <unistd.h>
5
6 int open(const char *pathname, int flags);
7 int open(const char *pathname, int flags, mode_t mode);
8 int close(int fd);
```

## Файл

```
1  int fd = open("info.txt",
2              O_WRONLY|O_CREAT|O_TRUNC,
3              S_IRUSR|S_IWUSR);
4  if (fd < 0) {
5      perror("Hey! You could not open file:");
6      return 1;
7  }
8  if (write(fd, "Hello", 6) < 0) {
9      perror("Hey! You could not write to file:");
10     return 1;
11 }
12 if (close(fd) < 0) {
13     perror("Hey! You could not close the file:");
14     return 1;
15 }
```

O\_WRONLY, O\_CREAT, O\_TRUNC — открыть файла на запись; создать, если нет файла; начать запись файла с начала.

S\_IRUSR S\_IWUSR — дать права запустившему пользователю на

## Права доступа

```
1  l  -l
2  drwxr-xr-x  2 user user 4096 sep 23 23:33 Downloads/
3  -rw-rw-r--  1 user user 8950 sep  1 12:54 1.py
```

Тип файла: d или -.

Права доступа: чтение (r -), запись (w -), запуск (x -) для пользователя, группы пользователей, всех остальных.

Права доступа можно записать в 8-ричной системе счисления (r=4), запись (w=2), запуск (x=1):

```
1  755 Downloads/
2  664 1.py
```

## Изменить права доступа

```
1 chmod 744  
2 chmod u=rwx 1.py  
3 chmod go=rw- 1.py  
4  
5 chmod +x 1.py
```

```
1 chown user:user 1.py
```

## Файл

Фильтр потока ошибок

```
1 ./prog 1>/dev/null
```

Логирование потока ошибок

```
1 ./prog 2>log.txt
```

Убрать поток ошибок

```
1 ./prog 2>/dev/null
```

Убрать оба потока

```
1 ./prog 2>&1 1>/dev/null
```

## Работа с директориями

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 #include <fcntl.h>
4 #include <unistd.h>
5
6 int mkdir(const char *pathname, mode_t mode);
7 int mkdir(const char *pathname);
8 int chdir(const char *pathname);
9 int rmdir(const char *pathname);
10 char *getcwd(char *buf, size_t size);
```

```
1 #include <sys/types.h>
2 #include <dirent.h>
3
4 DIR *opendir(const char *pathname);
5 struct dirent *readdir(DIR *dir);
6 int closedir(DIR *);
```

## Пример

```
1  #include <sys/types.h>
2  #include <dirent.h>
3
4  ...
5      int n_files = 0, n_dirs = 0;
6      char dir[] = "/home/user/";
7      DIR* d = opendir(dir);
8      if (d == NULL)
9          err(1, "opendir");
10
11     while (1) {
12         errno = 0;
13         struct dirent* entry = readdir(d);
14         if (errno != 0) {
15             err(1, "readdir");
16             if (entry == NULL)
17                 break;
18             puts(entry->d_name);
```

## Пример

```
1         if (entry->d_type == DT_DIR) {  
2             if (entry->d_name[0] == '.')  
3                 continue;  
4             n_dirs++;  
5         }  
6         if (entry->d_type == DT_REG)  
7             n_files++;  
8     }  
9     closedir(d);  
10    ...
```



## Идентификатор процесса pid

Посмотреть топ процессы

```
1 top
```

1	PID	USER	VIRT	%CPU	%MEM	TIME+	COMMAND
2	17800	alen	4508	100,0	0,0	0:07.04	prog03
3	17802	alen	52604	12,5	0,1	0:00.02	top
4	1275	alen	3837112	6,2	10,6	59:14.75	gnome-sh
5	16965	alen	2302848	6,2	6,9	1:31.32	firefox

Предварительно запустили висящий процесс с именем prog03.

## Идентификатор процесса pid

```
1 #include <unistd.h>
2
3 pid_t getpid(void);
4 pid_t getppid(void);
```

Родительский процесс — тот кто создал (по умолчанию init — процесс с id 1).

## prog03.c

```
1  #include <unistd.h>
2  #include <stdio.h>
3
4  int main(int argc, char **argv) {
5      pid_t pid = getpid();
6      printf("%u\n", pid);
7      while(1) {
8      }
9      return 0;
10 }
```

## Создание дочернего процесса

```
1 #include <sys/types.h>
2 #include <unistd.h>
3
4 pid_t fork(void);
```

Отличия: результат возврата fork (в дочке – ноль, в родителе – pid  
дочки).

Копируются: сегмент кода и сегменты данных.

Сохраняются: дескрипторы.

pid=17997

retpid=fork(); retpid==17998

pid=17998

retpid==0

## prog04.c

```
1  #include <sys/types.h>
2  #include <unistd.h>
3  #include <stdio.h>
4
5  int main(int argc, char **argv) {
6      pid_t pid = fork();
7      if (pid == 0) {
8          printf("I am child, pid: %u\n",
9              getpid());
10     } else {
11         printf("I am parent, pid: %u\n",
12             getpid());
13     }
14     while (1) {
15     }
16     return 0;
17 }
```

## prog04.c

Вывод:

```
1 I am parent, pid: 17997
2 I am child, pid: 17998
```

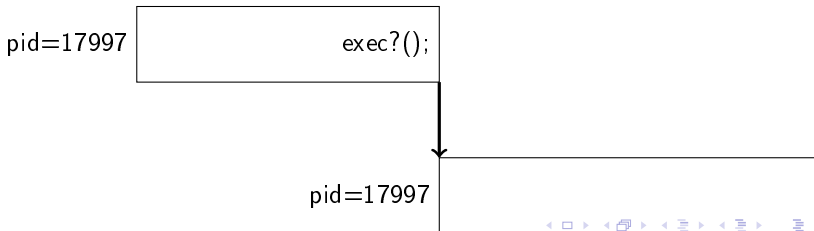
Результат top:

1	PID	USER	VIRT	%CPU	%MEM	TIME+	COMMAND
2	17997	alen	4508	100,0	0,0	0:15.20	prog04
3	17998	alen	4508	100,5	0,0	0:15.19	prog04

## Замена процесса exec

```
1 #include <unistd.h>
2 int execl(const char *path,
3           const char *arg,..., NULL);
4 int execlp(const char *file,
5           const char *arg,..., NULL);
6 int execv(const char *path, char *const argv[]);
7 int execvp(const char *file, char *const argv[]);
```

Заменяются: сегмент код, сегмент данных. Сохраняются:  
дескрипторы. Суффикс p: ищет в bin и /usr/bin.



## prog05.c

```
1 #include <unistd.h>
2 #include <stdio.h>
3 int main(int argc, char **argv) {
4     if (execlp("ls", "ls", "-l", NULL) < 0) {
5         perror("ls failed");
6         return 1;
7     }
8     return 0;
9 }
```

Запускается команда `ls`. В качестве аргументов передается: `ls` и `-l`.



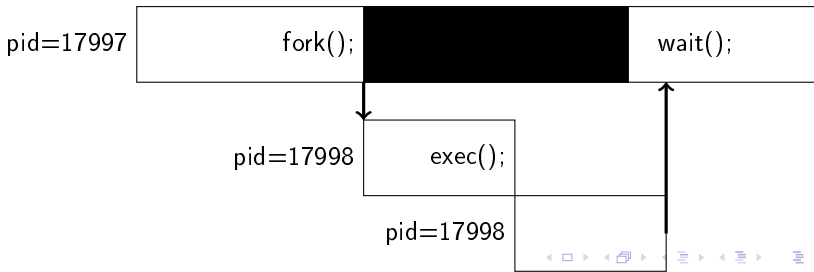
## prog06.c

```
1  #include <unistd.h>
2  #include <stdio.h>
3  int main(int argc, char **argv) {
4      char cmd[] = "ls";
5      char arg1[] = "-l";
6      char arg2[] = "-a";
7      char *arg_vec[4] = {cmd, arg1, arg2, NULL};
8      if (execvp(cmd, arg_vec) < 0) {
9          perror("ls failed");
10         return 1;
11     }
12     return 0;
13 }
```

arg\_vec — массив из четырех указателей типа char \* (то есть массив из 4 строк).

## Классическая схема запуска другого процесса

```
1  if (fork() > 0){ /* parent*/  
2      wait(NULL);  
3  } else { /* child */  
4      if (exec?(cmd, ...) < 0) {  
5          perror("exec_failed");  
6          return 1;  
7      }  
8  }
```



## write

```
1  .text
2  .global main
3  main:
4      mov $1, %rax //write
5
6      mov $1, %rdi    //write(1, ., .)
7      mov $ptr, %rsi  //write(., ptr, .)
8      mov $7, %rdx    //write(., ., 7)
9      syscall
10
11     ret
12
13 .data
14 ptr:
15     .ascii "Hello\n"
```

```
1 gcc prog.S -o prog -no-pie && ./prog
```

## Почитать

По системным вызовам

<https://ejudge.ru/study/3sem/unix.shtml>

По ассемблеру (факультативно) [http://blog.rchapman.org/posts/Linux\\_System\\_Call\\_Table\\_for\\_x86\\_64/](http://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/)