2.1 Системный вызов fork(). Системные вызовы семейства exec(). Примеры использования.

int fork () - создание нового процесса.

Дочерний процесс, является точной копией *родительского* процесса. **Различия м/у этими процессами:**

- Дочернему процессу присваивается уникальный идентификатор PID;
- Идентификаторы родительского проц. PPID у этих проц. различны;
- Дочерний проц. получает собственную копию и, в частности, собственные файловые дескрипторы, хотя он разделяет те же записи файловой таблицы.
-Дочерний проц. свободен от сигналов, ожидающих доставки;
-Временная статистика выполнения проц. в режиме ядра и задачи для дочернего проц. обнуляется.
-Блокировки памяти и записей, установленные родительским проц., потомком не наследуются.
- Значение, возвращаемое системным вызовом *fork()* различно для родителя и потомка (род. – PID потомка; дочерний процесс – 0, ошибка -1).

```
#include <sys/types>
 #include <unistd.h>
pid_t fork(void);
main(){
  pid_t pid;
  pid=fork();
if(pid==-1){printf("error");exit;}
if(pid==0){//потомок} else{//родитель} }
```

```
#include <unistd.h>
int execl(char *name, char *arg0, ... /*NULL*/);
int execv(char *name, char *argv[]);
int execle(char *name, char *arg0, ... /*,NULL, char
*envp[]*/);
int execve(char *name, char *arv[], char *envp[]);
int execlp(char *name, char *arg0, ... /*NULL*/);
int execvp(char *name, char *argv[]);
```

Вызов exec происходит таким образом, что переданная в качестве аргумента программа загружается в память вместо старой, которая вызвала exec. Старой программе больше не доступны сегменты памяти, которые перезаписаны новой программой.

Суффиксы l, v, p, e в именах функций определяют

- l (список). Аргументы командной строки передаются в форме списка arg0, arg1.... argn, NULL.
- v (vector). Последний аргумент (argv [n]) должен быть указателем NULL;
- p (path). Обозначенный по имени файл ищется не только в текущем каталоге, но и в каталогах, определенных переменной среды PATH;
- e (среда). Ф-ия ожидает список переменных среды в виде вектора (envp []) и не использует текущей среды.

Все формы системного вызова exec превращают вызвавший процесс в новый процесс, который строится из обычного выполняемого файла, называемого в дальнейшем новым выполняемым файлом. Выполняемый файл состоит из заголовка [см. a.out(4)], сегмента команд (.text) и данных. Данные состоят из инициализированной (.data) и неинициализированной (.bss) частей. Если системный вызов exec закончился успешно, то он не может вернуть управление, так как вызвавший процесс уже заменен новым процессом. Возврат из системного вызова exec свидетельствует об ошибке. В таком случае результат равен -1, а переменной errno присваивается код ошибки.

Начнем с примера для execl(). Пусть используется следующая программа:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  int i=0;
  printf("%s\n",argv[0]);
  printf("Программа запущена и получила строку : ");
  while(argv[++i] != NULL)
  printf("%s ",argv[i]);
  return 0;
}
```

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, int *argv[])
{
  printf("Будет выполнена программа %s...\n\n",
        argv[0]);
  printf("Выполняется %s", argv[0]);
  execl("hello"," ","Hello", "World!", NULL);
  return 0;
}
```

Эта программа выводит на экран строку, переданную ей в качестве аргумента. Пусть она называется hello. Она будет вызвана из другой программы с помощью функции execl(). Код вызывающей программы приведен справа.

В строке execl() аргументы указаны в виде списка. Доступ к ним также осуществляется последовательно.

2.2 Системные вызовы по управлению ФС. Примеры использования.

creat() Назначение: создание нового или изменения его атрибутов. **INCLUDE:** #include<fcntl.h>
Использование: int fd= **creat** (const char *path, mode_t mode);
ОПИСАНИЕ: аргумент path определяет имя файла в ФС, а mode — устанавливаемые права доступа к файлу. При этом выполняется ряд правил:

- Если идентификатор группы (GID) создаваемого файла не совпадает с эффективным идентификатором группы (EGID) или идентификатором одной из дополнительных групп процесса, бит SGID аргумента mode очищается (если он был установлен).
- Очищаются все биты, установленные в маске процесса umask.
- Очищается флаг Sticky bit.

Если файл существует, то он опустошается (размер становится = 0), а режим доступа и владелец не изменяются.
Результат: При успешном завершении результат равен неотриц. целому числу - дескриптору файла; в случае ошибки возвращается -1, а переменной errno присваивается код ошибки.
creat() ≡open(path, O_WRONLY | O_CREAT | O_TRUNC, mode);

OPEN(). Назначение: открыть файл для чтения или записи. **INCLUDE:** #include<fcntl.h>
Использование: int fd= **open** (const char *path, int oflag [, mode_t mode]);
ОПИСАНИЕ: аргумент *path* явл-ся указателем на маршрутное имя файла. Системный вызов open открывает дескриптор для указанного файла и устанавливает флаги статуса файла в соответствии со значением аргумента *oflag*.
1) O_RDONLY - Открыть только на чтение.
2) O_WRONLY - Открыть только на запись.
3) O_RDWR - Открыть на чтение/запись.
4) O_NDELAY - Этот флаг может воздействовать на последующие операции чтения и записи
5) O_APPEND - Перед каждой операцией записи устанавливать указатель текущей позиции на конец файла.
6) O_SYNC - Все записи в файл, а также соответствующие им изменения в метаданных файла будут сохранены на диске до возврата из вызова write().
7) O_CREAT - Если файл существует, то флаг игнорируется. В противном случае идентиф. владельца и группы создаваемого файла устанавливаются равными, соответственно, действующим идентиф. пользователя и группы процесса, а младшие 12 бит значения режима доступа к файлу устанавливаются равными значению аргумента mode, модифицированному как в creat().
8) O_TRUNC - Если файл существует, то он опустошается (размер становится = 0), а режим доступа и владелец не изменяются.
9) O_EXCL - Если установлены оба флага O_EXCL и O_CREAT, то системный вызов open завершается неудачей, если файл уже существует.
10) O_NOCTTY - Если указанный файл - терминал, не позволяет ему стать управляющим терминалом
11) O_NONBLOCK - Изменяет режим выполнения операций *read(2)* и *write(2)* для этого файла на неблокируемый. При невозможности произвести запись или чтение, например, если отсутствуют данные, соответствующие вызовы завершатся с ошибкой EAGAIN
Результат: При успешном завершении рез-том служит дескриптор файла; в случае ошибки возвращается -1, а переменной errno присваивается код ошибки. Возвращается наименьший из доступных дескрипторов.

READ( ) / READV ( ). Назначение: чтение данных из файла. **Использование:**
ssize_t read(int fields, void * buf, size_t nbyte); ssize_t readv(int fields, struct iovec *iov, int iovcnt);
ОПИСАНИЕ: аргумент *fildes* - дескриптор файла, полученный после выполнения системных вызовов creat(), open(), dup(), fcntl() или pipe(). Системный вызов read пытается прочитать *nbyte* байт из файла, ассоциированного с дескриптором *fildes*, в буфер, указателем на который является аргумент *buf*.
Для устройств, допускающих позиционирование, системный вызов read выполняет чтение из файла, начиная с указателя текущей позиции, ассоциированного с дескриптором fildes. После завершения чтения указатель текущей позиции файла увеличивается на количество прочитанных байт.

При попытке чтения из обычного файла с установленным флагом учета блокировки и при наличии блокировки на запись (другим процессом) того сегмента файла, который должен быть прочитан, в зависимости от значения флага O_NDELAY системный вызов read ведет себя следующим образом:
1. Если установлен флаг O_NDELAY, то возвр.знач. -1, а переменной errno присваивается код ош. EAGAIN.
2. Если флаг O_NDELAY не установлен, то читающий процесс откладывается до снятия блокировки.

При попытке чтения из пустого канала:
1. Если установлен флаг O_NDELAY, то системный вызов read возвращает значение 0.
2. Если не установлен флаг O_NDELAY, то читающий процесс откладывается до тех пор, пока данные не будут записаны в файл, или пока файл не перестанет быть открытым на запись.

При попытке чтения из файла, ассоциированного с терминалом, когда нет данных, предназначенных для чтения:
1. Если установлен флаг O_NDELAY, то возвращается значение 0.
2. Если не установлен флаг O_NDELAY, то читающий процесс откладывается до тех пор, пока данные не появятся.

При попытке чтения из файла, ассоциированного с потоком, в котором нет данных:
1. Если установлен флаг O_NDELAY, то возвр. знач. -1, а переменной errno присваивается код ош.EAGAIN.
2. Если не установлен флаг O_NDELAY, то читающий процесс откладывается до тех пор, пока данные не появятся.

Результат: При успешном завершении рез-т равен неотриц. целому числу – кол-ву реально прочитанных байт; Если текущая позиция совпадала с концом файла, результат будет = 0. В случае ошибки возвращается -1, а переменной errno присваивается код ошибки.

WRITE( ) / WRITEV( ). Назначение: запись в файл **Использование:**
ssize_t write (int fields, void * buf, size_t nbyte); ssize_t writev(int fields, struct iovec *iov, int iovcnt);
ОПИСАНИЕ: аргумент *fildes* - дескриптор файла, полученный после выполнения системных вызовов creat(2), open(2), dup(2), fcntl(2) или pipe(2). Системный вызов write пытается записать *nbyte* байт из буфера, на который указывает аргумент *buf*, в файл, ассоциированный с дескриптором *fildes*.
Запись в обычный файл блокируется, если установлен флаг учета блокировки и тот сегмент файла, в который производится попытка записи, заблокирован другим процессом. В этом случае, если не установлен флаг O_NDELAY, записывающий процесс откладывается до снятия блокировки сегмента.
При попытке записать большее кол-во байт, чем позволяет максим. размер файла или наличие свободного пр-ва на устройстве, записывается столько байт, сколько возможно. При установленном флаге O_NDELAY запись в полный канал приводит к возврату значения 0. Если флаг O_NDELAY не установлен, запись в полный канал задерживается до тех пор, пока не освободится пространство для записи.
Результат: При успешном завершении результат равен неотрицательному целому числу - количеству реально записанных байт; в случае ошибки возвращается -1, а переменной errno присваивается код ошибки

DUP( ) / DUP2( ) . Назначение: дублирование дескриптора открытого файла. **Использование:** int dup (int fildes);
ОПИСАНИЕ: аргумент *fildes* - дескриптор файла, полученный после выполнения системных вызовов creat, open, dup, fcntl и pipe. Возвращается наименьший из доступных дескрипторов.
Функция dup2() делает то же самое, однако позволяет указать номер файлового дескриптора, который требуется получить после дублирования: *int dup2 (int fildes, int fildes2); fildes* - дублируется, *fildes2* - новый дескриптор. Если fildes2 уже занят, то сначала выполняется close(fildes2);
Результат: В случае успешного завершения ф-ии возвращается новый файловый дескриптор, св-ва которого идентичны св-вам дескриптора fildes. Оба указывают на один и тот же файл, одно и то же смещение, начиная с которого будет производиться следующая операция чтения или записи (файловый указатель), и определяют один и тот же режим работы с файлом. В случае ошибки возвращается -1, а переменной errno присваивается код ошибки.

CLOSE(). **Назначение:** разрывает связь м/у файловым дескриптором и открытым файлом.
Использование: int close (int fildes);
ОПИСАНИЕ: аргумент *fildes* - дескриптор файла, полученный в рез-тате выполнения системных вызовов creat, open, dup, fcntl или pipe. Последний вызов close для потока, связанного с дескриптором fildes, приводит к ликвидации потока.
Результат: При успешном завершении рез-тат = 0; в случае ошибки возвращается -1, а переменной errno присваивается код ошибки.

LSEEK ( ). Назначение: передвижение указателя чтения/записи
Использование: off_t lseek (int fildes, off_t offset, int whence); в нек. источниках off_t => long
ОПИСАНИЕ: аргумент *fildes* - дескриптор файла, полученный после выполнения сист. вызовов creat, open, dup или fcntl. *lseek* устанавливает указатель текущей позиции файла, ассоциированного с дескриптором *fildes*, следующим образом, в зависимости от значения аргумента *whence*:
SEEK_CUR (1) Указатель смещается на offset байт от текущего положения

SEEK_END (2) Указатель смещается на offset байт от конца файла
SEEK_SET (0) Указатель устанавливается равным offset
Результат: При успешном завершении рез-том служит неотрицательное целое число - указатель текущей позиции в файле; в случае ошибки возвращается -1, а переменной errno присваивается код ошибки

PIPE( ). Назначение: создание однонаправленного (симплексного) канала (также называемого анонимным каналом) обмена данными м/у двумя родственными процессами. **Использование:** int pipe (int fields[2]);
ОПИСАНИЕ: pipe создает механизм ввода/вывода, называемый каналом, и возвращает 2 дескриптора файла fildes[0] и fildes[1]. Дескриптор fildes[0] открыт на чтение, fildes[1] - на запись.
Канал буферизует до 5120 байт данных; запись в него большего кол-ва инф-ии без считывания приведет к блокированию пишущего процесса. Посредством дескриптора fildes[0] инф-ия читается в том же порядке, в каком она записывалась с помощью дескриптора fildes[1].
Результат: При успешном завершении рез-тат = 0; в случае ошибки возвращается -1, а переменной errno присваивается код ошибки.

2.3 Временные параметры файлов. Удаление открытых файлов. Файл с дырой

Индексный дескриптор любой UNIX-подобной системы традиционно хранит три временные метки, показывающие, когда над индексным дескриптором или файлом в последний раз была совершена та или иная операция.

mtime - **modification time** - время последней модификации (изменения) файла

atime - **access time** - время последнего доступа к файлу

ctime - **change time** - время последнего изменения атрибутов файла (данных которые хранятся в **inode-области**) Изменяется тогда когда изменяются права доступа к файлу (командой **chmod**), изменяется владелец файла (команда **chown**), создаются жесткие ссылки на файл(команда **ln**).

Все значения времени, связанные с файлом (время доступа, модификации данных и метаданных) хранятся в секундах, прошедших с 0 часов января 1970 года. Заметим, что информация о времени создания файла отсутствует. просмотреть можно с помощью системных вызовов stat():

```
# include <sys/types.h>
# include <sys/stat.h>
#include <time.h>
main (int arc, char *argv[])
{       struct stat s;
        lstat (argv[1], &s);
printf ("atime=%s", ctime(&s.st_atime));
printf ("mtime=%s", ctime(&s.st_mtime));
printf ("ctime=%s", ctime(&s.st_ctime));   }
```

CLOSE(). Назначение: разрывает связь между файловым дескриптором и открытым файлом

Использование: *int close (int fildes);*

ОПИСАНИЕ: аргумент *fildes* - дескриптор файла, полученный в результате выполнения системных вызовов creat, open, dup, fcntl или pipe. Последний вызов close для потока, связанного с дескриптором fildes, приводит к ликвидации потока.

Результат: При успешном завершении результат = 0; в случае ошибки возвращается -1, а переменной errno присваивается код ошибки.

Файл с дырой

Разрежённый файл — файл, в котором последовательности нулевых байтов заменены на инф-ию об этих последовательностях (список дыр).

Дыра— последовательность нулевых байт внутри файла, не записанная на диск. Информация о дырах (смещение от начала файла в байтах и количество байт) хранится в метаданных ФС.

Преимущества:

- экономия дискового пространства. Использование разрежённых файлов считается одним из способов сжатия данных на уровне ФС;
- отсутствие временных затрат на запись нулевых байт;
- увеличение срока службы запоминающих устройств.

Недостатки:

- накладные расходы на работу со списком дыр;
- фрагментация файла при частой записи данных в дыры;
- невозможность записи данных в дыры при отсутствии свободного места на диске.

```
int fd, nbytes;
char string[]="Hello! \n";
fd = creat("test",777);
nbytes=write(fd,string,strlen(string));
lseek(fd,100,1);
nbytes=write(fd,string,strlen(string));
```

2.4 Системный вызов fcntl(2). Блокировки файлов. Примеры использования.

FCNTL(2) -управление файлами. **Использование:** *int fcntl(int fildes, int cmd, ...);*
ОПИСАНИЕ: Системный вызов fcntl выполняет управляющие операции над открытыми файлами. Аргумент *fildes* - дескриптор открытого файла; *cmd* может принимать след. значения, определяющие выполняемую операцию, а возможный третий аргумент зависит от конкретного действия:

```
F_DUPFD    0 /* Скопировать дескриптор файла */
F_GETFD    1 /* Получить флаги файла с данным дескриптором */
F_SETFD    2 /* Установить флаги файла с данным дескриптором */
F_GETFL    3 /* Получить флаги файла */
F_SETFL    4 /* Установить флаги файла */
F_GETLK    5 /* Получить состояние блокировки файла */
F_SETLK    6 /* Установить блокировку файла */
F_SETLKW   7 /* Установить блокировку файла и ждать */
F_CHKFL    8 /* Проверить допустимость изменений флагов файла */
F_ALLOCSP 10 /* Зарезервирован */
F_FREESP  11 /* Отмена резерва */
```

Рез-тат: При успешном завершении в зависимости от операции *cmd* возвращаются следующие значения:

- **F_DUPFD** Новый дескриптор файла.
- **F_GETFD** Значение флага (определен только младший бит).
- **F_SETFD** Значение, отличное от -1.
- **F_GETFL** Значение флагов статуса файла.
- **F_SETFL** Значение, отличное от -1.
- **F_GETLK** Значение, отличное от -1.
- **F_SETLK** Значение, отличное от -1.
- **F_SETLKW** Значение, отличное от -1.

В случае ошибки возвращается -1, а переменной errno присваивается код ошибки.

LOCKF(3C) - блокировка сегментов файла. **СИНТАКСИС:** *int lockf (int fildes, int function, long size)*
ОПИСАНИЕ: позволяет блокировать отдельные сегменты файла. Учитывать ли блокировку при записи, определяется режимом доступа к файлу. Если другие процессы попытаются блокировать уже заблокированный фрагмент, они либо получат в ответ код ошибки, либо будут ждать освобождения ресурса. При завершении процесса все блокировки, установленные им, удаляются.
Аргумент *fildes* - дескриптор открытого файла. Чтобы функция lockf завершилась успешно, файл должен быть открыт с правом записи (O_WRONLY или O_RDWR). Аргумент *function* - значение, задающее выполняемые действия:

```
F_ULOCK 0 /* Разблокировать ранее блокированный сегмент */
F_LOCK  1 /* Заблокировать сегмент */
F_TLOCK 2 /* Проверить и заблокировать сегмент */
F_TEST  3 /* Проверить сегмент */
```

Другие значения function зарезервированы для будущих расширений и приводят к ошибке, если не реализованы.
Аргумент *size* – кол-во последовательных байт файла, которые должны быть заблокированы или разблокированы. Если *size* = 0, блокируется (разблокируется) сегмент от текущей позиции до конца файла. Блокируемый сегмент не обязан существовать в файле, допустима блокировка областей за концом файла.
Результат: При успешном завершении рез-тат = 0; в случае ошибки возвращается -1, а переменной errno присваивается код ошибки.

FCNTL(5) - флаги управления файлами. Структура контроля за блокировкой сегмента файла, информация передается пользователем системе

```
struct flock {
 short l_type; /* тип блокировки */
 short l_whence; /*начальное смещение*/
 long  l_start; /*относительное смещение*/
 long  l_len;   /* Если 0, то до конца файла */
 short l_sysid; /* Длина блокируемой записи.
Возвращается по запросу F_GETLK */
 short l_pid;   /* Идентификатор процесса,
пост.блокировку. Возвращается по запросу
F_GETLK */   };
```

Типы блокировок сегмента файла

```
 F_RDLCK 01 /* Блокировка на чтение */
 F_WRLCK 02 /* Блокировка на запись
*/
     F_UNLCK 03 /* Снятие блокировки
*/
```

Традиционно архитектура файловой подсистемы UNIX разрешает нескольким процессам одновременный доступ к файлу для чтения и записи.
Есть возможность устанавливать блокировки. По умолчанию блокирование является ***рекомендательным*** (advisory lock). Т.е. кооперативно работающие процессы могут руководствоваться созданными блокировками, однако ядро не запрещает чтение или запись в заблокированный участок файла. При работе с рекомендательными блокировками процесс должен явно проверять их наличие с помощью функций fcntl(2) и *lockf(3C).*
Правила блокирования такие, что может быть установлено несколько блокирований для чтения на конкретный байт файла, при этом в установке блокирования для записи на этот байт будет отказано. Напротив, блокирование для записи на конкретный байт должно быть единственным, при этом в установке блокирования для чтения будет отказано.

Приведем фрагмент программы, использующей возможность блокирования записей:

```
  struct flock lock;
/*заполним описание lock с целью блокирования всего файла для записи*/
lock.l_type =F_WRLCK;
lock.l_start=0;
lock.whence=SEEK_SET;
lock.len=0;
/*Заблокируем файл. Если блокирования, препятствующие данной операции, уже существуют - ждем их снятия*/
fcntl(fd,SETLKW,&lock);
/*Записи данных в файл ничего не помешает*/
write(fd,record,sizeof(record));
/*Снимем блокирование*/
lock.l_type =F_UNLCK;
fcntl(fd,SETLKW,&lock);
```

В отличие от рекомендательного в UNIX существует *обязательное блокирование* (mandatory lock), при котором ограничение на доступ к записям файла накладывается самим ядром. Реализация обязательных блокировок может быть различной.

2.5 Системные вызовы signal( ) и sigaction( ). Примеры использования.

include <signal.h> *void (* signal(int sig, void(*disp)(int))) (int);*
sig - определяет сигнал, диспозицию которого нужно изменить; *disp* - определяет новую диспозицию сигнала:
1) определенная пользователем функция обработчик
2) SIG_DFL - выполнение действия по умолчанию
3) SIG_IGN - сигнал необходимо игнорировать (не все сигналы возможно игнорировать).

В случае успешного завершения signal() возвращает предыдущую диспозицию.
Использование функции signal() подразумевает семантику устаревших или ненадежных сигналов. Процесс при этом имеет весьма слабые возможности управления сигналами. а) процесс не может заблокировать сигнал т.е. отложить получение сигнала на момент выполнения критического участка кода б) каждый раз при получении сигнала его диспозиция устанавливается в действие по умолчанию

```
void trap2(int sig) {
        long secs;
        secs=time(0);
        fprintf(stderr,"\n########### This is signal !!!!! T=%d  My PID=%d ########\n", secs, getpid());
} main()
{....
        signal(SIGUSR1, trap2);
        signal(SIGUSR2, SIG_IGN);
....}
```

Вместо функции signal() стандарт POSIX.1 определяет функцию ***sigaction(),*** позволяющую установить диспозицию сигналов, узнать ее текущее значение, или сделать и то и другое одновременно. Надежные сигналы.
#include <signal.h> *int sigaction(int sig, const struct sigaction *act, struct sigaction *oact);*
также принимает номер сигнала (кроме SIGKILL и SIGSTOP). Второй аргумент – новое описание для сигнала, через третий возвращается старое значение.

Структура sigaction имеет слудующие поля:
void (*sa_handler) () - обработчик сигнала sig
void (*sa_sigaction) (int,siginfo_t *, void *) - обработчик сигнала sig при установленном флаге SA_SIGINFO
sigset_t sa_mask - маска сигналов
int sa_flags – флаги:
SA_NOCLDSTOP - Если *sig* равен **SIGCHLD**, то уведомление об остановке доч. процесса не будет получено (т.е. в тех случаях, когда доч. процесс получает сигнал **SIGSTOP**, **SIGTSTP**, **SIGTTIN** или **SIGTTOU**).
SA_ONESHOT или **SA_RESETHAND** - Восстановить поведение сигнала после одного вызова обработчика.
SA_ONSTACK - Вызвать обработчик сигнала в дополнительном стеке сигналов, предоставленном **sigaltstack**(2). Если дополнительный стек недоступен, то будет использован стек по умолчанию.
SA_RESTART - Поведение должно соответствовать семантике сигналов BSD и позволять некоторым системным вызовам работать, в то время как идет обработка сигналов.
SA_NOMASK or **SA_NODEFER** - Не препятствовать получению сигнала при его обработке.
SA_SIGINFO - Обработчик сигнала требует 3-х аргументов, а не одного. В этом случае надо использовать параметр *sa_sigaction* вместо *sa_handler*.

Пример отыскивает инф-ию относительно текущего действия для SIGINT без замены этого действия.

```
struct sigaction query_action;
  if (sigaction (SIGINT, NULL, &query_action) < 0)
        /* sigaction возвращает -1 в случае ошибки. */
  else if (query_action.sa_handler == SIG_DFL)
        /* SIGINT обработан заданным по умолчанию, фатальным способом. */
  else if (query_action.sa_handler == SIG_IGN)
        / * SIGINT игнорируется. * /
  else  /* Определенный программистом обработчик сигнала. */
```

2.6 Особенности сигналов SIGKILL, SIGSTOP, SIGINT, SIGQUIT, SIGHUP, SIGCHLD, SIGFPE.

Название	Код	Действие по умолчанию	Описание	Тип
SIGKILL	9	**Завершение**	**Безусловное завершение**	**Управление**
SIGSTOP	23	**Остановка процесса**	**Остановка выполнения процесса**	**Управление**
SIGINT	2	Завершение	Сигнал прерывания (Ctrl-C) с терминала	Управление
SIGQUIT	3	Завершение с дампом памяти	Сигнал «Quit» с терминала (Ctrl-\)	Управление
SIGHUP	1	Завершение	Закрытие терминала	Уведомление
SIGCHLD	18	Игнорируется	Дочерний процесс завершен или остановлен	Уведомление
SIGFPE	8	Завершение с дампом памяти	Ошибочная арифметическая операция	Исключение

В POSIX-системах **SIGCHLD** — сигнал, посылаемый при изменении статуса дочернего процесса (завершен, приостановлен или возобновлен).

Сигнал **SIGHUP** посылается:
- При прерывании соединения на последовательной линии, программам, запущенным с терминала, подключенного к ней, часто из-за того, что пользователь отсоединяет свой модем от телефонной или выделенной линии. ОС определяет разрыв соединения по исчезновению сигнала «несущая» (англ. *carrier detect, DCD*) последовательного порта.
- При закрытии псевдо- или виртуальных терминалов, которые используются на современных системах вместо аппаратных терминалов.
- Утилитой или функцией kill, с консоли или из скрипта/утилиты для управления *демоном*, для выполнения предусмотренного действия (обычно — перечитывания конфигурации и переинициализации).
- Для предотвращения завершения SIGHUP *стандартных* программ и утилит, существует утилита nohup («префикс» для программы в командной строке). nohup настраивает игнорирование SIGHUP, после чего запускает программу с аргументами в фоновом режиме с перенаправлением вывода в файл nohup.out в текущем или домашнем каталоге пользователя.

Будучи посланным процессу, **SIGKILL** вызывает его **немедленное** завершение. В отличие от SIGTERM или SIGINT этот сигнал не может быть перехвачен или проигнорирован, а процесс, получивший его не имеет возможности выполнить какие-либо действия перед своим завершением.
- процесс-зомби нельзя завершить SIGKILL, т.к. зомби уже завершен и не может принимать сигналов. Зомби ожидает, что родительский процесс считает код завершения с помощью системного вызова wait().
- Процессы в состоянии блокировки (например, при ожидании ввода-вывода) не смогут завершиться SIGKILL, пока ОС не вернет их в норм. состояние (при наступлении ожидаемого события или ошибке).
- Процесс *init* является особым случаем — он не получает от ОС сигналов, которые он не хочет обрабатывать, и, следовательно, может игнорировать SIGKILL.

В POSIX-системах, **SIGQUIT** — сигнал, для остановки процесса пользователем, комбинацией «quit» на терминале. Этот сигнал также указывает, что система должна выполнить дамп памяти для процесса.

В POSIX-системах, **SIGSTOP** — сигнал, посылаемый для принудительной приостановки выполнения процесса. Для возобновления выполнения используется сигнал SIGCONT. В отличие от сигнала SIGTSTP, SIGSTOP не может быть обработан программой или проигнорирован.

2.7 Синхрониз. процессов с использ. над. и ненадежных сигналов. Пр-ры программ

Сигналы - простейший способ межпроцессного взаимодействия. Предназначены для информирования процесса о наступлении какого-либо события. Причины отправки:

- По инициативе процесса – системный вызов **int kill(pid_t *pid*, int *sig*);**

Если значение *pid* является положительным, сигнал *sig* посылается процессу с идентификатором *pid*.
Если *pid* = 0, то *sig* посылается каждому процессу, который входит в группу текущего процесса.
Если *pid* = -1, то *sig* посылается каждому процессу, за исключением процесса с номером 1 (init), но есть нюансы, которые описываются ниже.
Если *pid* меньше чем -1, то *sig* посылается каждому процессу, который входит в группу процесса *-pid*.
Если *sig* = 0, то никакой сигнал не посылается, а только выполняется проверка на ошибку.
В случае успеха, возвращается ноль. При ошибке, возвращается -1

- По инициативе ядра:

1. Нажатие на управляющем терминале определенных клавиш (драйвер терминала),
2. Аппаратные особые ситуации
3. Программные состояния (будильники)

Доставка сигнала: диспозиция (SIG_IGN, SIG_DFL, пользовательская функция)
void (* signal(int sig, void(*disp)(int))) (int); где функция-обработчик *void disp(int sig);*
sig - сигнал, диспозицию которого нужно изменить; *disp* - новую диспозицию сигнала. В случае успешного завершения signal() возвращает предыдущую диспозицию.
Использование функции signal() подразумевает семантику устаревших или ненадежных сигналов:
а) процесс не может заблокировать сигнал т.е. отложить получение сигнала на момент выполнения критического участка кода
б) каждый раз при получении сигнала его диспозиция устанавливается в действие по умолчанию

Вместо функции signal() стандарт POSIX.1 определяет функцию **sigaction()** - надежные сигналы.
int sigaction(int sig, const struct sigaction *act, struct sigaction *oact);
также принимает номер сигнала (кроме SIGKILL и SIGSTOP). Второй аргумент - новое описание для сигнала, через третий возвращается старое значение. Структура sigaction имеет следующие поля:
void (*sa_handler) () - обработчик сигнала sig
void (*sa_sigaction) (int,siginfo_t *, void *) - обработчик сигнала sig при установленном флаге SA_SIGINFO
sigset_t sa_mask - маска сигналов
int sa_flags – флаги:
SA_NOCLDSTOP - Если *sig* = **SIGCHLD**, то уведомление об остановке доч. процесса не будет получено
SA_ONESHOT или **SA_RESETHAND** - Восстановить поведение сигнала после одного вызова обработчика.
SA_ONSTACK - Вызвать обработчик сигнала в дополнительном стеке сигналов, предоставленном **sigaltstack**(2). Если дополнительный стек недоступен, то будет использован стек по умолчанию.
SA_RESTART -Поведение должно соответствовать семантике сигналов BSD и позволять некоторым системным вызовам работать, в то время как идет обработка сигналов.
SA_NOMASK or **SA_NODEFER** - Не препятствовать получению сигнала при его обработке.
SA_SIGINFO -Обработчик сигнала требует 3-х аргументов, а не одного. В этом случае надо использовать параметр *sa_sigaction* вместо *sa_handler*.
Дополнительные возможности:

- Блокирование доставки сигнала
- Рекурсивная обработка
- Получение дополнительной информации вместе с сигналом
- Управление блокирующими системными вызовами

Библиотечные функции, для работы с сигналами:
int sigemptyset(sigset_t **set*); инициализирует набор сигналов, указанный в *set*, и "очищает" его от всех сигналов.
int sigfillset(sigset_t **set*); полностью инициализирует набор *set*, в котором содержатся все сигналы.
int sigaddset(sigset_t **set*, int *signum*); и **int sigdelset(sigset_t **set*, int *signum*);** добавляют сигналы *signum* к *set* и удаляют эти сигналы из набора соответственно.
int sigismember(const sigset_t **set*, int *signum*); проверяет, является ли *signum* членом *set*.
sigemptyset, **sigfullset**, **sigaddset** и **sigdelset** при удачном завершении возвращают 0 и -1 при ошибках.
sigismember возвращает 1, если *signum* является членом *set*; возвращает 0, если *signum* не является членом, а -1 возвращается при ошибках.

Управление маской сигналов
int sigprocmask(int *how*, const sigset_t **set*, sigset_t **oldset*); - исп-ся для того, чтобы изменить список блокированных в данный момент сигналов. Работа ф-ии зависит от значения параметра *how* след. образом:
SIG_BLOCK – Набор блокируемых сигналов - объединение текущего набора и аргумента *set*.
SIG_UNBLOCK – Сигналы, устанавливаемое значение битов которых равно *set*, удаляются из списка блокируемых сигналов. Допускается разблокировать незаблокированные сигналы.
SIG_SETMASK – Набор блокируемых сигналов приравнивается к аргументу *set*.
Если значение поля *oldset* не равно 0, то предыдущее значение маски сигналов записывается в *oldset*.

Работа с сигналами
Системный вызов **int sigpending(sigset_t **set*);** позволяет определить наличие ожидающих сигналов (полученных заблокированных сигналов). Маска ожидающих сигналов помещается в *set*.
Системный вызов **int sigsuspend(const sigset_t **mask*);** временно изменяет значение маски блокировки сигналов процесса на указанное в *mask*, и затем приостанавливает работу процесса до получения соответствующего сигнала. **int pause();** - ожидание сигналов

Функции **sigprocmask** и **sigpending** возвращают 0 при удачном завершении работы функции и -1 при ошибке. Функция **sigsuspend** всегда возвращает -1, обычно с кодом ошибки **EINTR**.

Доставка и обработка сигнала
Вызов функции ядра issig() от имени процесса приводит к действию по умолчанию или вызову ф-ии sendsig(), запускающей обработчик сигнала. При этом обработчик выполняется в режиме задачи.
- При возврате из режима ядра в режим задачи после обработки системного вызова или прерывания
- Перед переходом процесса в состояние сна с приоритетом допускающим прерывание сигналом
- После пробуждения от сна с приоритетом, допускающим прерывание сигналом

Пример:
```
void trap2(int sig) {
    long secs;
    secs=time(0);
    fprintf(stderr,"\n############ This is signal !!!!! T=%d  My PID=%d ########\n", secs, getpid());
}
main()
{....
    signal(SIGUSR1, trap2);
    signal(SIGUSR2, SIG_IGN);
....}
```

2.8 Системные вызовы для работы с каналами. Примеры использования

Каналы обеспечивают передачу информации в виде потока байтов без сохранения границ сообщений.
• неименованные (каналы)
• именованные (FIFO-файлы)

pipe(2) - создание однонаправленного (симплексного) канала для обмена данными м/у 2-мя родственными процессами. *int pipe (int fields[2]);*

Системный вызов pipe создает механизм ввода/вывода, называемый каналом, и возвращает 2 дескриптора файла fildes[0] и fildes[1]. Дескриптор fildes[0] открыт на чтение, дескриптор fildes[1] - на запись.
Канал буферизует до 5120 байт данных; запись в него большего кол-ва инф-ии без считывания приведет к блокированию пишущего процесса. Посредством дескриптора fildes[0] инф-ия читается в том же порядке, в каком она записывалась с помощью дескриптора fildes[1].
При успешном завершении рез-тат = 0; в случае ошибки возвращается -1, а переменной errno присваивается код ошибки.

int pipe2(int fd[2], int flags) - Если *flags* = 0, то **pipe2**() выполняет то же что и **pipe**(). **Флаги:**
O_CLOEXEC- Устанавливает флаг close-on-exec (**FD_CLOEXEC**) для двух новых открытых файловых дескрипторов.
O_DIRECT (начиная с Linux 3.4) - Создаёт канал, в котором ввод-вывод выполняется в «пакетном» режиме. Каждый **write**(2) в канал рассматривается как отдельный пакет, а **read**(2) из канала читает один пакет за раз. Старые ядра, которые не поддерживают этот флаг, возвращают ошибку **EINVAL**.
O_NONBLOCK (O_NDELAY) - Устанавливает флаг состояния файла **O_NONBLOCK** для двух новых открытых файловых дескрипторов.

Для создания FIFO используется системный вызов *mknod(2)*: **int mknod(char *pathname, int mode, int dev);** где *pathname* - имя файла в ФС (имя FIFO), *mode* — флаги владения, прав доступа и т.д. *dev* — при создании FIFO игнорируется.
После создания FIFO м.б. открыт на запись и чтение, причем запись и чтение могут происходить в разных независимых процессах. В качестве пр-ра приведем простейший пример приложения клиент- сервер, использующего FIFO для обмена данными. Клиент посылает серверу сообщение "Здравствуй, Мир!", а сервер выводит это сообщение на терминал.

сервер

```
#include <sys/types.h>
#include <sys/stat.h>
#define FIFO "fifo.1"
#define MAXBUFF 80
main() {
 int readfd, n;
 char buff[MAXBUFF]; /* буфер для чтения данных из
FIFO */
 /* Создадим специальный файл FIFO с открытыми для
всех правами доступа на чтение и запись */
 if (mknod(FIFO, S_IFIFO | 0666, 0) < 0) {
  printf("Невозможно создать FIFO\n");
  exit(1); }
 /* Получим доступ к FIFO */
 if ((readfd = open(FIFO, O_RDONLY)) < 0) {
  printf("Невозможно открыть FIFO\n");
  exit(1); }
 /* Прочитаем сообщение ("Здравствуй, Мир!") и
выведем его на экран */
 while ((n = read(readfd, buff, MAXBUFF)) > 0)
  if {write(1, buff, n) != n) {
   printf("Ошибка выводан");
   exit(1);  }
 /* Закроем FIFO, удаление FIFO - дело клиента */
 close(readfd);
 exit(0);}
```

клиент

```
#include <sys/types.h>
#include <sys/stat.h>
/* Соглашение об имени FIFO */
#define FIFO "fifo.1"
main() {
 int writefd, n;
 /* Получим доступ к FIFO */
 if ((writefd = open(FIFO, O_WRONLY)) < 0) {
  printf("Невозможно открыть FIFO\n");
  exit(1);
 }
 /* Передадим сообщение серверу FIFO */
 if (write(writefd, "Здравствуй, Мир!\n", 18) != 18) {
  printf("Ошибка записи \n");
  exit(1);
 }
 /* Закроем FIFO */
 close(writefd);
 /* Удалим FIFO */
 if (unlink(FIFO) < 0) {
  printf("Невозможно удалить FIFO\n");
  exit(1);
 }
 exit(0);
}
```

2.9 Организация процессов-демонов. Примеры

Демоны – специальные системные процессы, работающие длительное время, не связанные с терминалами. Управление обычно осуществляется при помощи сигналов. В крайнем случае убить и запустить заново с другими параметрами.

init – самый главный процесс в системе. pid=1 Является отцом всех осиротевших процессов. Активно работает в момент начальной загрузки системы и при переходе с уровня на уровень. В остальное время спит. Смерть init приводит к состоянию kernel panic.

Последовательность шагов при создании демона:

Снять ассоциацию с управляющим терминалом (Демон не должен получать SIGHUP от терминала)
Закрыть все открытые файлы (Демон может выводить сообщения только через syslog)
Сменить текущий каталог на корневой (Демон не должен мешать размонтированию ФС)
Выполнить fork, родительский процесс exit, код в тело дочернего процесса (ppid = 1)

Известные демоны: inetd – сетевой суперсервер; crond – демон расписания; sendmail – демон почтовой службы; httpd – демон web-сервера; syslogd – демон системного журнала

Скелет программы-демона:

```
#include <stdio.h>
#include <syslog.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/param.h>
#include <sys/resource.h>

main (int argc, char** argv)
{
int fd;
struct rlimit flim;
        if (getpid()!=1)
        {       signal(SIGTTOU,SIG_IGN);        посылается при попытке вывода на управл. терминал
                signal(SIGTTIN,SIG_IGN);        посылается при попытке чтения с управл. терминала
                signal(SIGTSTP,SIG_IGN);        посылается с терминала при оставноке вып-ия процесса
                if (fork()!=0)
                        exit(0);
                setsid();  }
        getrlimit(RLIMIT_NOFILE,&flim);         ограничение использования ресурсов
        for(fd=0;fd<flim.rlim_max;fd++)
        {  close(fd);    }
        chdir("/");
        openlog("My daemon", LOG_PID|LOG_CONS, LOG_DAEMON);   устанавливает связь с программой, ведущей лог
        syslog(LOG_INFO,"Daemon started");      создает сообщение для журнала
        closelog();   }
```

2.10 Очереди сообщений. Примеры использования.

Очереди сообщений являются составной частью UNIX System V, они обслуживаются ОС, размещаются в адресном пространстве ядра и являются разделяемым системным ресурсом.
Процессы могут записывать и считывать сообщения из различных очередей. Процесс, пославший сообщение в очередь, может не ожидать чтения этого сообщения каким-либо другим процессом. Данная возможность позволяет процессам обмениваться структурированными данными, имеющими следующие атрибуты:

- Тип сообщения (позволяет мультиплексировать сообщения в одной очереди)
- Длина данных сообщения в байтах (может быть нулевой)
- Собственно, данные (если длина ненулевая, могут быть структурированными)

Очередь сообщений хранится в виде **внутреннего однонаправленного связанного списка** в адресном пр-ве ядра. Для каждой очереди ядро создает заголовок очереди (msqid_ds), где содержится инф-ия о правах доступа к очереди (msg_perm), ее текущем состоянии (msg_cbytes – число байтов и msg_qnum— число сообщений в очереди), а также указатели на первое (msg_first) и последнее (msg_last) сообщения, хранящиеся в виде связанного списка. Каждый элемент этого списка является отдельным сообщением.

Для создания новой очереди сообщений или для доступа к существующей используется системный вызов msgget(2): *int msgget(key_t key, int msgflag);*
Функция возвращает дескриптор объекта-очереди, либо -1 в случае ошибки.
Подобно файловому дескриптору, этот идентификатор используется процессом для работы с очередью сообщений. В частности, процесс может:

- Помещать в очередь сообщения с помощью функции msgsnd(2)
- Получать сообщения определенного типа из очереди с помощью функции msgrcv(2)
- Управлять сообщениями с помощью функции msgctl(2)

*int msgsnd (int msgid, const void *msgp, size_t msgsz, int msgflg);*
*int msgrcv (int msgid, void *msgp, size_t msgsz, long msgtyp, int msgflg);*

Здесь *msgid* - дескриптор объекта, *msgp* указывает на буфер, содержащий тип сообщения и его данные, размер которого равен *msgsz* байт. Буфер имеет следующие поля:

- long msgtype - тип сообщения
- char msgtext[] - данные сообщения

Если msgtyp = 0, ф-ия msgrcv() получит первое сообщение из очереди. Если величина msgtyp выше 0, будет получено первое сообщение указанного типа. Если msgtyp меньше 0, функция msgrcv() получит сообщение с минимальным значением типа, меньше или равного абсолютному значению msgtyp.
Стандартный порядок получения сообщ. аналогичен принципу FIFO — сообщения получаются в порядке их записи. Однако используя тип, напр-р, для назначения приоритета сообщений, этот порядок легко изменить.
Пример приложения "Здравствуй, Мир!" использующего сообщения:

Файл описания mesg.h

```
#define MAXBUFF 80
#define PERM 0666
typedef struct our msgbuf {
long mtype;
char buff[MAXBUFF];
} Message;
```

Пример программы клиента

```
#include <sys/types.h>
ttinclude <sys/ipc.h>
ttinclude <sys/msg.h>
#include "mesg.h"
main()
{ Message message;
key_t key;
int msgid, length ;
message.mtype=1L;
if ( (key=ftok("server",'A') ) < 0) {
        printf("it is impossible to obtain the key\n"); exit(1);  }
if ((msgid=msgget(key,0)) < 0) {
        printf("it is impossible to gain access to the queue\n");
exit(1);  }
if ( (length=sprintf(message.buff,"Здравствуй, Мир!")) <0 ) {
        printf("error copying to the buffer\n"); exit(1);  }
if (msgsnd(msgid, (void*)&message, length, 0) !=0){
        printf("error writing to queue\n"); exit(1);  }
if (msgctl(msgid,IPC_RMID,0) < 0) {
        printf("failed to delete the queue\n");exit(1); }
exit(0);  }
```

++

Пример программы сервера

```
#include <sys/types.h>
ttinclude <sys/ipc.h>
ttinclude <sys/msg.h>
#include "mesg.h"
main()
{  Message message;
key_t key;
int msgid,length,n;
if ( (key=ftok("server",'A') ) < 0) {
   printf("it is impossible to obtain the key\n"); exit(1);  }
message.mtype=1L;
if ((msdig=msgget(key,PERM | IPC_CREAT) <0 ){
   printf("it is impossible to create a queue\n"); exit(1);  }
n=msgrcv(msgid,
&message,sizeof(message),message.mtype,0);
if (n>0) {
        if(write(1,message.buff, n ) !=n) {
                printf("error output\n");exit(1);  } }
else { printf("read error\n");exit(1);}
exit(0);  }
```

2.11 Семафоры. Примеры использования

Для синхронизации доступа нескольких процессов к разделяемым ресурсам, используются семафоры. Являясь одной из форм IPC, семафоры не предназначены для обмена большими объемами данных, как в случае FIFO или очередей сообщений. Вместо этого, они разрешают или запрещают процессу использование того или иного разделяемого ресурса.
Для нормальной работы необходимо обеспечить выполнение следующих условий:
1. Значение семафора должно быть доступно различным процессам. Поэтому семафор находится не в адресном пространстве процесса, а в адресном пространстве ядра.
2. Операция проверки и изменения значения семафора должна быть реализована в виде одной атомарной по отношению к другим процессам (т. е. непрерываемой другими процессами) операции. В противном случае возможна ситуация, когда после проверки значения семафора выполнение процесса будет прервано другим процессом, который в свою очередь проверит семафор и изменит его значение. Единственным способом гарантировать атомарность критических участков операций яв-ся выполнение этих операций в режиме ядра.

Для каждой группы семафоров ядро поддерживает структуру данных **semid_ds**, включающую след. поля:
struct ipc_perm sem_perm Описание прав доступа
struct sem *sem_base Указатель на первый элемент массива семафоров
ushort sem_nsems Число семафоров в группе
time_t sem_otime Время последней операции
time_t sem_ctime Время последнего изменения

Значение конкретного семафора из набора хранится во внутренней структуре sem:
ushort semval Значение семафора
pid_t sempid Идентификатор процесса, выполнившего последнюю операцию над семафором
ushort semncnt Число процессов, ожидающих увеличения значения семафора
ushort semzcnt Число процессов, ожидающих обнуления семафора

Для получения доступа к семафору (и для его создания, если он не существует) используется системный вызов **semget(2)**: ***int semget(key_t key, int nsems, int semmflag);***
nsems - число семафоров в группе; *semflag* - права доступа к семафору и флажки для его создания **(IPC_CREAT, IPC_EXCL).**
После получения дескриптора объекта процесс может производить операции над семафором. Для этого используется системный вызов **semop(2): int semop (int semid, struct sembuf *semop, size_t nops)**
Второй аргумент - указатель на структуру данных, определяющую операции, которые требуется произвести над семафором с дескриптором *semid*. Операций может быть несколько, и их число указывается в последнем аргументе *nops.*
Каждый элемент набора операций имеет вид:
struct sembuf {
short sem num /*номер семафора в группе*/ **short sem_op;** /*операция*/ **short sem flg;** /*флаги операции*/ **}**

UNIX допускает 3 возможные операции над семафором, определяемые полем **semop**:
1. Если величина **semop** положительна, то текущее значение семафора увеличивается на эту величину.
2. Если значение **semop** равно нулю, процесс ожидает, пока семафор не обнулится.
3. Если величина **semop** отрицательна, процесс ожидает, пока значение семафора не станет большим или равным абсолютной величине **semop**. Затем абсолютная величина **semop** вычитается из значения семафора.

ПР-Р: рассмотрим 2 случая использования бинарного семафора (т.е. значения могут принимать только 0 и 1). В первом пр-ре значение 0 - разрешающее, а 1 запирает некоторый разделяемый ресурс (файл, разделяемая память), ассоциированный с семафором. Определим операции, запирающие ресурс и освобождающие его:
static struct = sembuf sop_lock[2] = { 0, 0, 0
0, 1,0 **};** /*ожидать обнуления семафора*/ /*затем увел. знач. семафора на 1 */
static struct sembuf sop unlock [1] = { 0,-1,0 **};** /*обнулить значение семафора*/
Итак, для запирания ресурса процесс производит вызов: **semop(semid, &sop_lock[0], 2);**
обеспечивающий атомарное выполнение двух операций:
1. Ожидание доступности ресурса. В случае, если ресурс уже занят (значение семафора = 1), выполнение процесса будет приостановлено до освобождения ресурса (значение семафора = 0).
2. Запирание ресурса. Значение семафора устанавливается равным 1.
Для освобождения ресурса процесс должен произвести вызов: **semop(semid, &sop_unlock[0],1);**

который уменьшит текущее значение семафора (равное 1) на 1 и оно станет = 0, что соответствует освобождению ресурса. Если какой-либо из процессов ожидает ресурса (т. е. произвел вызов операции **sop_lock**), он будет "разбужен" системой, и сможет в свою очередь запереть ресурс и работать с ним.

Во втором примере изменим трактовку значений семафора: значению 1 семафора – доступность некоторого ассоциированного с семафором ресурса, а 0 — недоступность. В этом случае содержание операций несколько изменится.
static struct sembuf sop_lock[2] = { 0, -1, 0, **};** /*ожидать разрешающего сигнала (1),затем обнулить семафор*/
static struct sembuf sop_unlock [1] = { 0, 1, 0 **};** /*увеличить значение семафора на 1*/
Процесс запирает ресурс вызовом: **semoop(semid, &sop_lock[0], 1);**
а освобождает: **semoop(semid, &sop_unlock[0], 1);**

Во втором случае операции получились проще (по крайней мере их код стал компактнее), однако этот подход имеет потенциальную опасность: при создании семафора, его значения устанавливаются равными 0, и во втором случае он сразу же запирает ресурс. Для преодоления данной ситуации процесс, первым создавший семафор, должен вызвать операцию **sop_unlock**, однако в этом случае процесс инициализации семафора перестанет быть атомарным и может быть прерван другим процессом, который, в свою очередь, изменит значение семафора. В итоге, значение семафора станет равным 2, что повредит нормальной работе с разделяемым ресурсом.

2.12 Разделяемые сегменты памяти. Примеры использования

Примерный сценарий работы с разделяемой памятью выглядит следующим образом:
1. Сервер получает доступ к разделяемой памяти, используя семафор.
2. Сервер производит запись данных в разделяемую память.
3. После завершения записи сервер освобождает разделяемую память с помощью семафора.
4. Клиент получает доступ к разделяемой памяти, запирая ресурс с помощью семафора.
5. Клиент производит чтение данных из разделяемой памяти и освобождает ее, используя семафор.

Для каждой области разделяемой памяти, ядро поддерживает структуру данных **shmid_ds** основными полями которой являются:

```
struct shmid_ds {
struct ipc_perm shm_perm; /* права доступа */
int shm_segsz; /* размеры сегмента (в байтах) */
time_t shm_atime; /* время последней привязки */
time_t shm_dtime; /* время последней отвязки */
time_t shm_ctime; /* время последнего изменения */
unsigned short shm_cpid; /* pid создателя */
unsigned short shm_lpid; /* pid последнего пользователя сегмента */
short shm_nattch; /* номер текущей привязки */
/* следующее носит частный характер */
unsigned short shm_npages; /* размеры сегмента (в страницах) */
unsigned long *shm_pages; /* массив указателей на $frames -> S$ */
struct vm_area_struct *attaches; /* дескрипторы для привязок */
```

Чтобы создать новый разделяемый сегмент памяти или получить доступ к уже существующему, используется системный вызов **shmget().** *int shmget (key_t key, int size, int shmflg);*
RETURNS: идентификатор разделяемого сегмента памяти в случае успеха, -1 в случае ошибки:
Первый аргумент - это значение ключа. Это значение ключа затем сравнивается с существующими значениями, которые находятся внутри ядра для других разделяемых сегментов памяти.

Операция открытия или получения доступа зависит от содержания аргумента **shmflg.**
IPC_CREAT -Создает сегмент, если он еще не существует в ядре.
IPC_EXCL - При использовании совместно с **IPC_CREAT** приводит к ош., если сегмент уже существует.
Если исп-ся один **IPC_CREAT**, **shmget()** возвращает идентиф. для вновь созданного сегмента или идентиф. для сегмента, который уже существует с тем же значением ключа. Если вместе с **IPC_CREAT** используется **IPC_EXCL**, тогда создается новый сегмент или если сегмент уже сущ., вызов "проваливается" с -1.

Для работы с разделяемой памятью (чтение и запись) необходимо сначала присоединить (attach) область вызовом shmat(2): *char * shmat(int shmid, char * shmaddr, int shmflag);*
возвращает адрес начала области в адресн. пр-ве процесса размером *size* заданным предшествующем вызовом shmget(). В этом адресн. пр-ве взаимодействующие процессы могут размещать требуемые структуры данных для обмена информацией. Правила получения этого адреса следующие:
1. Если аргумент *shmaddr* нулевой, то система самостоятельно выбирает адрес.
2. Если *shmaddr* отличен от 0, знач. возвращаемого адреса зависит от наличия флага SHM_RND в shmflag
   - Если SHM_RND не установлен, система присоединяет разделяемую память к указанному shmaddr адресу.
   - Если SHM_RND устан., система присоединяет разд. память к адресу, полученному округлением в меньшую сторону shmaddr до некоторой определенной величины SHMLBA.

По умолч. разд. память присоединяется с правами на чтение и запись. Права меняются флагом в shmaddr.
Окончив работу с разд. памятью, процесс отключает (detach) область вызовом int shmdt (char * shmaddr);

Системный вызов **shmctl** позволяет выполнять операции управления разделяемыми сегментами памяти.
*int shmctl (int shmid, int cmd, struct shmid_ds *buf);*
RETURNS: 0 в случае успеха, -1 в случае ошибки
Употребляемые значения команд следующие:
IPC_STAT - Берет структуру **shmid_ds** для сегмента и сохраняет ее по адресу, указанному buf-ом.
IPC_SET - Присвоить соответствующим полям shmid значения, находящиеся в buf:
IPC_RMID -Удалить из системы shmid, ликвидировать сегмент и ассоциированную с ним структуру данных.
SHM_LOCK -Удержать в памяти разделяемый сегмент shmid.
SHM_UNLOCK -Освободить разделяемой сегмент памяти shmid.

Пример не из рабочевского!
Давайте создадим функцию-переходник для обнаружения или создания разделяемого сегмента памяти:

```
int open_segment( key_t keyval, int segsize )
{ int shmid;
if((shmid = shmget( keyval, segsize, IPC_CREAT | 0660 )) == -1)
{ return(-1); }
return(shmid); }
```

Рассмотрим функцию-переходник, которая по корректному идентификатору сегмента возвращает адрес привязки сегмента:

```
char *attach_segment( int shmid )
{ return(shmat(shmid, 0, 0)); }
```

2.13 Файлы, отображаемые в память. Примеры использования

Системный вызов *mmap(2)* предоставляет механизм доступа к файлам, альтернативный вызовам *read(2)* и *write(2)*. С помощью этого вызова процесс имеет возм-ть отобразить участки файла в собственное адр.пр-во. После этого данные файла м.б. получены или записаны путем чтения или записи в память. Функция *mmap(2)* определяется след. образом: *caddr_t mmap(caddr_t addr, size_t len, int prot, int flags, int fildes, off_t off);*

Вызов задает отображение *len* байтов файла с дескриптором *fildes*, начиная со смещения *off*, в область памяти со стартовым адресом *addr*. Перед вызовом *mmap(2)* файл должен быть открыт с помощью функции *open(2)*. Аргумент *prot* опред-ет права доступа к области памяти, кот. должны соответствовать правам доступа к файлу, указанным в системном вызове *open(2)*.

Возможные значения аргумента prot и соответствующие им права доступа к файлу

Значение аргумента prot	**Описание**	**Права доступа к файлу**
PROT_READ	Область доступна для чтения	r
PROT_WRITE	Область доступна для записи	w
PROT_EXEC	Область доступна для исполнения	x
PROT_NONE	Область недоступна	-

Обычно значение *addr* задается = 0, что позволяет ОС самостоятельно выбрать виртуальный адрес начала области отображения. В любом случае, при успешном завершении возвращаемое сист. вызовом значение определяет действительное расположение области памяти.

ОС округляет значение *len* до следующей страницы виртуальной памяти.
При обращении к участку памяти, лежащему за пределами файла, ядро отправит процессу сигнал SIGBUS. Несмотря на то что область памяти может превышать фактический размер файла, процесс не имеет возможности изменить его размер.

Аргумент *flags* определяет дополнительные особенности управления областью памяти.

Значение аргумента flags	**Описание**
MAP SHARED	Область памяти может совместно использоваться несколькими процессами
MAP PRIVATE	Область памяти используется только вызывающим процессом
MAP_FIXED	Требует выделения памяти, начиная точно с адреса addr
MAP_NORESERVE	Не требует резервирования области свопинга

Процесс также может явно снять отображение с помощью вызова *munmap(2)*. Закрытие файла не приводит к снятию отображения. Снятие отображения непосредственно не влияет на отображаемый файл, т. е. содержимое страниц области отображения не будет немедленно записано на диск. Обновление файла производится ядром согласно алгоритмам управления виртуальной памятью. В то же время в ряде систем существует функция *msync(3C)*, которая позволяет синхронизировать обновление памяти с обновлением файла на диске.

В качестве примера приведем упрощенную версию утилиты *cp(1)*, копирующую один файл в другой с использованием отображения файла в память.

```
main(int argc, char *argv[]) {
 int fd_src, fd_dst;
 caddr_t addr_src, addr_dst;
 struct stat filestat;
 /* Первый аргумент - исходный файл, второй - целевой */
 fd_dst=open(argv[2], O_RDWR | O_CREAT);
 /* Определим размер исходного файла */
 fstat(fd_src, &filestat);
 /* Сделаем размер целевого файла равным исходному */
 lseek(fd_dst, filestat.st_size - 1, SEEK_SET);
 /* Зададим отображение */
 addr_src=mmap((caddr_t)0, filestat.st_size,
  PROT_READ, MAP_SHARED, fd_src, 0);
 addr_dst=mmap((caddr_t)0, filestat.st_size,
  PROT_READ | PROT_WRITE, MAP_SHARED,
fd_dst, 0);
 /* Копируем области памяти */
 memcpy(addr_dst, addr_src, filestat.st_size);
 exit(0);
}
```

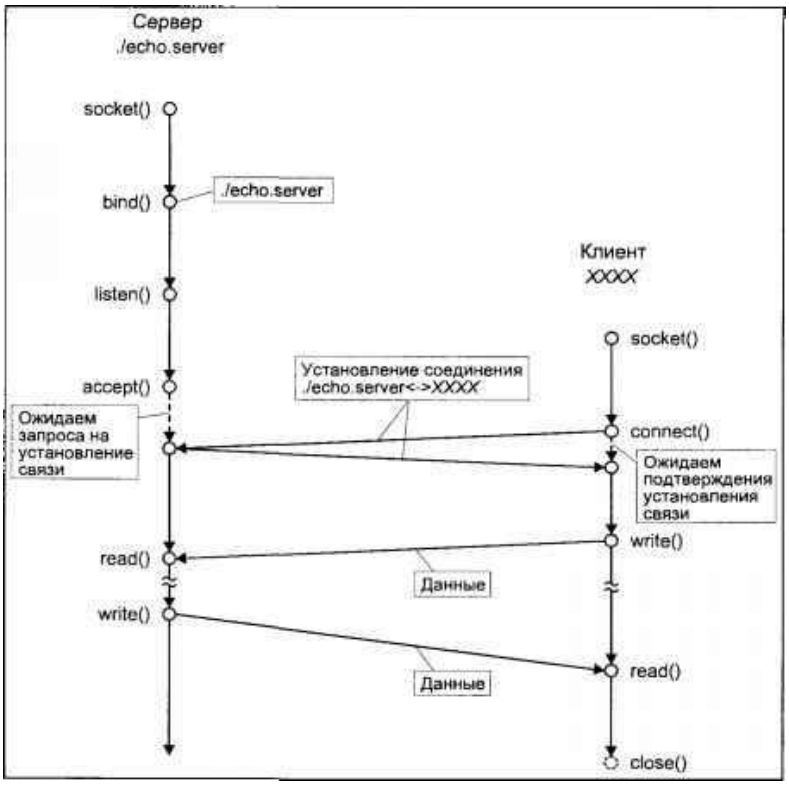
2.14 Потоковые сокеты. Принципы создания клиент-серверных приложений

Сокеты являются коммуникационным интерфейсом взаимодействующих процессов.
Для создания сокета используется системный вызов *socket(2)*, имеющий следующий вид:
int socket(int domain, int type, int protocol);
Аргумент *domain* – коммуникационный домен, *type* — тип сокета (SOCK_STREAM - потоковые сокеты), а *protocol* — используемый протокол (м.б. не указан, т.е. = 0). В случае успеха систем. вызов возвращает полож. целое число, аналогичное файловому дескриптору, которое служит для адресации данного сокета в последующих вызовах.

Коммуникационный домен определяет семейство протоколов, допустимых в рамках данного домена. Возможные значения аргумента domain включают:

AF_UNIX/ PF_UNIX	Домен локального межпроцессного взаимодействия в пределах единой операционной системы UNIX. Внутренние протоколы.
AF_INET/ PF_INET	Домен взаимодействия процессов удаленных систем. Протоколы Internet (TCP/IP).
AF_NS	Домен взаимодействия процессов удаленных систем. Протоколы Xerox NS.

Домен может не поддерживать определенные типы сокетов. AF_UNIX И AF_INET поддерживают SOCK_STREAM
Также допустимы не все комбинации типа сокета и используемого коммуникационного протокола (если таковой явно указан в запросе). Так для домена AF_INET SOCK_STREAM - IPPROTO_TCP (TCP), (SOCK_DGRAM -UDP)
Иллюстрация взаимодействия м/у процессами при виртуальном коммуникационном канале **с предварительным установлением связи** приведена на рис.



Связывание может быть осуществлено с помощью системного вызова *bind(2)*: *int bind(int sockfd, struct sockaddr *localaddr, int addrlen);*

sockfd - дескриптором сокета, полученным при его создании; *localaddr* – локальный адрес, с которым необходимо связать сокет; параметр *addrlen* определяет размер адреса.

Адрес сокета зависит от коммуникационного домена, в рамках которого он определен. В общем случае адрес определяется следующим образом (в файле **<sys/socket.h>**):

```
struct sockaddr {
 u_short sa_family;
 char sa_data[14]; }
```

Поле *sa_family* – коммуникационный домен (семейство протоколов), *sa_data* — содержит собственно адрес, формат которого определен для каждого домена.
Например, для внутреннего домена UNIX адрес выглядит следующим образом (определен в **<sys/un.h>**):

```
struct sockaddr_un {
 short sun_family; /* ==AF_UNIX */
 char sun_path[108];  };
```

Для домена Internet (сем-во протоколов TCP/IP) исп-ся след. формат адреса (опр-н в файле **<netinet/in.h>**):

```
struct sockaddr_in {
 short sin_ family; /* ==AF_INET */
 u_short sin_port;
 struct in_addr sin_addr;
 char sin_zero[0];    }
```

Вызов *connect(2)* имеет вид: *int connect(int sockfd, struct sockaddr *servaddr, int addrlen);*
Предполагает создание вирт. канала, и исп-ся для предварительного установления связи м/у коммуникационными узлами. Локальный узел коммуникационного канала указывается дескриптором сокета sockfd, для которого система автоматически выбирает приемлемые значения локального адреса и процесса.

Удаленный узел определяется аргументом servaddr, который указывает на адрес сервера, а addrlen задает его длину.

Системный вызов *listen(2)* информирует систему, что сервер готов принимать запросы. Он имеет следующий вид: *int listen(int sockfd, int backlog);*
sockfd - сокет, который будет использоваться для получения запросов, *backlog* - максимальное число запросов на установление связи, которые могут ожидать обработки сервером.

Фактическую обработку запроса клиента на установление связи производит системный вызов *accept(2)*: *int accept(int sockfd, struct sockaddr *clntaddr, int* addrlen);*
accept(2) извлекает первый запрос из очереди и создает новый сокет, хар-ки которого не отличаются от сокета *sockfd*, и таким образом завершает создание виртуального канала со стороны сервера. Одновременно *accept(2)* возвращает пар-ры удаленного коммуникационного узла — адрес клиента *clntaddr* и его размер *addrlen*. Новый сокет используется для обслуживания созданного виртуального канала, а полученный адрес клиента исключает анонимность последнего.

Дальнейший типичный сценарий взаимодействия имеет вид:

```
        sockfd = socket(...);           Создать сокет
        bind(sockfd, ...);              Связать его с известным локальным адресом
        listen(sockfd, ...);            Организовать очередь запросов
        for(;;)
{       newsockfd = accept(sockfd, ...); Получить запрос
                if (fork() == 0) {              Породить дочерний процесс
                        close(sockfd);                  Дочерний процесс
                        ...
                        exit(0);
                } else
                        close(newsockfd);               Родительский процесс    }
```

В этом сценарии, в то время как дочерний процесс обеспечивает фактический обмен данными с клиентом, родительский процесс продолжает "прослушивать" поступающие запросы, порождая для каждого из них отдельный процесс-обработчик. Очередь позволяет буферизовать запросы на время, пока сервер завершает вызов *accept(2)* и затем создает дочерний процесс. Заметим, что новый сокет newsockfd, полученный в результате вызова *accept(2)*, адресует полностью определенный коммуникационный канал: протокол и полные адреса обоих узлов — клиента и сервера. Напротив, для сокета sockfd определена только локальная часть канала. Это позволяет серверу продолжать использовать sockfd для "прослушивания" последующих запросов.

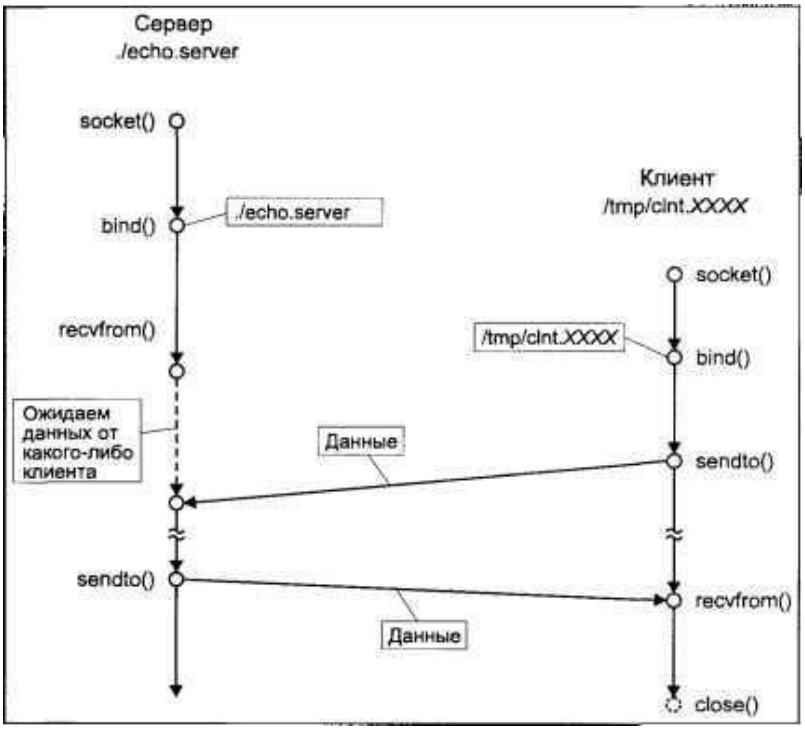
для сокетов потока при приеме и передаче данных могут быть использованы стандартные вызовы *read(2)* и *write(2)*

2.15 Дейтаграммные сокеты. Систем. вызовы для работы с дейтаграммными сокетами

Сокеты яв-ся коммуникационным интерфейсом взаимодействующих процессов. Для создания сокета используется системный вызов *socket(2)*: *int socket(int domain, int type, int protocol);*
domain - коммуникационный домен, *type* — тип сокета (SOCK_DGRAM), *protocol* — используемый протокол (может быть не указан, т.е. = 0). В случае успеха системный вызов возвращает полож. целое число, аналогичное файловому дескриптору, которое служит для адресации данного сокета в последующих вызовах. Коммуникационный домен определяет семейство протоколов (protocol family), допустимых в рамках данного домена. Возможные значения аргумента domain включают:

AF_UNIX/ PF_UNIX	Домен локального межпроцессного взаимодействия в пределах единой операционной системы UNIX. Внутренние протоколы.
AF_INET/ PF_INET	Домен взаимодействия процессов удаленных систем. Протоколы Internet (TCP/IP).
AF_NS	Домен взаимодействия процессов удаленных систем. Протоколы Xerox NS.

Домен может не поддерживать определенные типы сокетов. AF_UNIX И AF_INET поддерживают SOCK_ DGRAM
Также допустимы не все комбинации типа сокета и используемого коммуникационного протокола (если таковой явно указан в запросе). Так для домена AF_INET SOCK_ DGRAM использует IPPROTO_UDP (UDP)
Взаимодействие между процессами, основанное на датаграммах (**без предварительного установления соединения**)



Связывание м.б. осуществлено с помощью системного вызова *bind(2)*: *int bind(int sockfd, struct sockaddr *localaddr, int addrlen);*
sockfd - дескриптор сокета, полученный при его создании; *localaddr* - локальный адрес, с которым необходимо связать сокет; *addrlen* - размер адреса. Заметим, что речь идет о связывании с локальным адресом, в общем случае определяющим два параметра коммуникационного канала (коммуникационный узел): локальный адрес и локальный процесс.

Адрес сокета зависит от коммуникационного домена, в рамках которого он определен. В общем случае адрес определяется следующим образом (в файле **<sys/socket.h>**):

```
struct sockaddr {
 u_short sa_family;
 char sa_data[14];      }
```

Поле sa_family - коммуникационный домен (семейство протоколов), а sa_data —адрес, формат которого определен для каждого домена.
Например, для внутреннего домена UNIX адрес выглядит следующим образом (определен в **<sys/un.h>**):

```
struct sockaddr_un {
 short sun_family; /* ==AF_UNIX */
 char sun_path[108];  };
```

Для домена Internet (сем-во протоколов TCP/IP) исп-ся след. формат адреса (определен в файле **<netinet/in.h>**):

```
struct sockaddr_in {
 short sin_ family; /* ==AF_INET */
 u_short sin_port;
 struct in_addr sin_addr;
 char sin_zero[0];  }
```

При приеме и передаче данных сокеты датаграмм должны пользоваться специальными системными вызовами:

*int sendto(int s, const char *msg, int len, int flags, const struct sockaddr* toaddr, int tolen);*
*int recvfrom(int s, char *buf, int len, int flags, struct sockaddr* fromaddr, int* fromlen);*

Первый аргумент - дескриптор сокета, через который производится обмен данными. Аргумент *msg* содержит сообщение длиной *len*, которое должно быть передано по адресу *toaddr*, длина которого составляет *tolen* байтов. Аргумент *buf* представляет собой буфер, в который копируются полученные данные.

Параметр *flags* может принимать следующие значения:

MSG_OOB Передать или принять экстренные данные вместо обычных

MSG_PEEK Просмотреть данные, не удаляя их из системного буфера (последующие операции чтения получат те же данные)