1. Программа make.

make — утилита, автоматизирующая процесс преобразования файлов из одной формы в другую. Чаще всего это компиляция исходного кода в объектные файлы и последующая компоновка в исполняемые файлы или библиотеки.

Назначение– выполнение минимально необходимых действий по пересборке проекта.

Описание проекта и зависимостей файлов – makefile.

Файлы по умолчанию:

./makefile ./Makefile.

./s.makefile, SCCS/s.makefile, ./s.Makefile, SCCS/s.Makefile

Опции:

-f makefilename задает имя файла с описанием зависимостей

-i игнорирует коды завершения операций
-k при ошибке продолжает сборку других целей
-n печатает команды, но их не выполняет
-p печатает макросы на терминал
-q код возврата не 0, если что-то надо делать
-r очищает список суффиксов и не использует встроенные правила
-t обновляет время целей, но сборку не производит
-s не выводит выполняемые команды
MAKEFLAGS= опции по умолчанию.

2. Программа SCCS.

Эта система предназначена для контроля и документирования текстовых файлов путем создания некоторого аналога учетной ведомости.

Инструментальное средство SCCS представляет собой пакет программ, который может функционировать под управлением OS типа UNIX (в том числе, CONVEX-OS и OC System V), файловая система которых не поддерживает такой атрибут, как версия.

SCCS: source code control system
хранит историю изменений файла: s.файл

Программы/команды

admin создает файлы, изменяет параметры
delta вносит изменения в файл истории
get запрашивает файл для редактирования или компиляции
prs печатает файл с комментариями
rmdel удаляет внесенные изменения
sact показывает, кто работает с файлом
unget отказывается от намерения внести изменения
val проверяет историю изменений
what ищет подстановку по %Z%

3. Основные стандарты интерфейсов программирования ОС UNIX.

C89, C99, C11, C14, C17

C89

1989 г. – первый формальный стандарт

- прототипы функций
- битовые поля, объединения и т.п.
- стандартная библиотека
- функции с переменным числом аргументов, void, volatile …

В стандарте C89 определены 5 основных типов данных:

int — целые, float — с плавающей точкой, double — двойной точности, char — символьные данные, void — без значения. Остальные типы C (Си) основываются на них.

C99

- Массивы с размерностью, определяемой переменной
- комплексные числа (несовместимо с Си++)

В C99 было добавлено несколько новых возможностей, многие из которых уже были реализованы в качестве расширения в некоторых компиляторах.

- Встраиваемые функции (объявленные с ключевым словом inline).
- Место, в котором возможно объявление переменных, больше не ограничено глобальной областью видимости и началом составного оператора (блока).
- Несколько новых типов данных, включая long long int, дополнительные расширенные целые типы, явный логический тип данных, а также комплексный тип (complex) для представления комплексных чисел.
- Массивы переменной длины (variable-length arrays).
- Поддержка однострочных комментариев, начинающихся с //, как в BCPL или C++.
- Новые библиотечные функции, как, например, snprintf.
- Новые заголовочные файлы, такие как stdbool.h и inttypes.h.
- Типовые математические функции (tgmath.h).
- Улучшена поддержка стандарта IEEE 754-2008.
- Составные константы (например, стало возможным определять структуры прямо в вызове функции: function((struct point){4,2})).
- Новые инициализаторы для массивов и структур(int a[10] = { [0] = 100, [3] = 200}; struct mystruct {int a; int b; int c;} ob = { .c = 30, .a = 10 };).
- Поддержка вариативных макросов (макросов переменной арности).
- Смягчение (restrict) ограничений для более агрессивной оптимизации кода.

C11 C14

- Улучшенная и расширенная поддержка Unicode
- Поддержка многопоточности
- Добавлено больше макросов для получения характеристик чисел с плавающей точкой
- Спецификатор функции _Noreturn
- Выравнивание данных. Для этого в язык добавили спецификатор _Alignas, оператор alignof, функцию aligned_alloc и заголовочный файл <stdalign.h>
- проверка выхода за границы массивов
- анонимные структуры, вывод типа и др.

4. Статические библиотеки. Создание и использование.

Библиотеки статической компоновки lib*.a.*

Подключение библиотеки: -lимя
где файл библиотеки libимя.a.* или libимя.so.*

Создание статических библиотек
ar [опции] файл_архива[файл]

Опции:
-q добавитьфайлв конец архива
-d удалитьфайлиз архива
-t напечатать таблицу содержимого архива
-p напечатать содержимого указанных файлов или всего архива
-r заменить или добавить файлы
 -x извлечь из архива файл, если не указать– все.
-m переместить файл, используется вместе с –a, –b, –i.
-s обновить таблицу символов

Модификаторы
-a файл после этого файла
-b файл перед этим файлом
-i файл перед этим файлом
-u обновить файлы (вместе с –r).

В Linux после внесения изменений: ranlib файл_архива.a (для обновления__SYMDEF)

5. Разделяемые библиотеки. Создание библиотеки. Вызов функций при динамической компоновке.

Создание разделяемых библиотек
В исполняемом файле– только ссылка на библиотеку
Компилировать, указывая –fPIC position independent code
Объединить в библиотеку–shared –fPIC

Пример:
gcc –c –fPIC d1.c
gcc –c –fPIC d2.c
gcc –shared –fPIC –o libnew.sod1.o d2.o

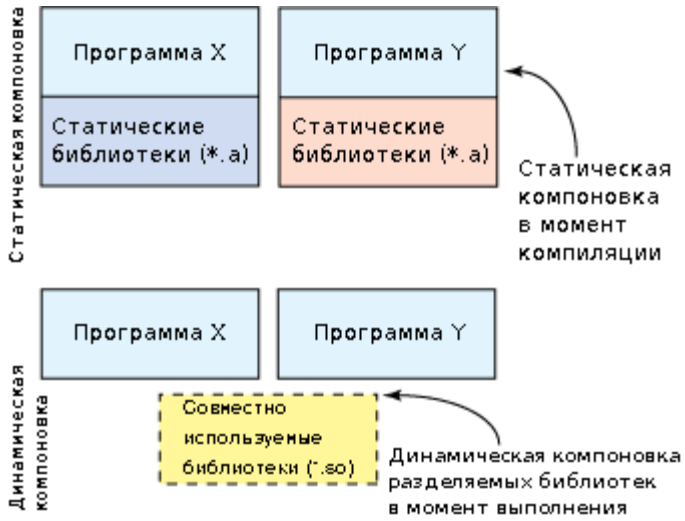
gcc a.c –o awlnew–L. –lnew
export LD_LIBRARY_PATH=.
./awlnew или
 gcc a.c –o awlnew–L. –lnew–Wl,-rpath,/export/home/user/project1

Зависимость библиотек друг от друга:	порядок указания важен?
статические	ДА
динамические	НЕТ

явный вызов функций библиотеки в процессе т. н. динамической загрузки. В этом случае программа явно загружает нужную библиотеку, а затем вызывает определенную библиотечную функцию. На этом методе обычно основан механизм загрузки подключаемых программных модулей – плагинов. Оба рассматриваемых способа показаны на рисунке 2.

Рисунок 2. Сравнение статической и динамической компоновки

Сравнение статической и динамической компоновки



Динамическаязагрузка и выгрузка (явнаякомпоновка)

Использовать: <dlfcn.h> libdl.so int dlclose(void *); char *dlerror(void); void *dlopen(const char *, int); флаги RTLD_LAZY, RTLD_NOW, RTLD_GLOBAL, RTLD_LOCAL void *dlsym(void *restrict, const char *restrict);

Пример использования(без обработкиошибок!): void *handle = dlopen(“libnew.so”, RTLD_LAZY); // if (handle ==0 … void (*test)() = dlsym(handle, “newfunction”); // if (test ==0 … (*test)(); dlclose(handle);

gcc –rdynamic –o foofoo.c-ldl

6. Статическая и динамическая настройка программы по системным ограничениям и стандартам.

Настройка программы настандарт

Проверка версии реализованного в системе стандарта
<unistd.h>
#if …==…
_POSIX_SOURCE //POSIX.1-1990
_POSIX_C_SOURCE //POSIX.1-1990 иPOSIX.1-1992
_POSIX_C_SOURCE==199309L // POSIX.1b-1993
_POSIX_C_SOURCE==199506L // POSIX.1b-1996
_POSIX_C_SOURCE==200112L // POSIX.1-2001
_XOPEN_SOURCE == 700 //XPG4, SUSv4
_XOPEN_SOURCE_EXTENDED == 1

_POSIX_VERSION == 200809L //POSIX.1-2008
_POSIX2_VERSION == 200809L
_XOPEN_VERSION == 700 //X/Open PortabilityGuide

7. Стандартная библиотека ANSI C.

Содержимое и форма стандартной библиотеки С задается Стандартом ANSI/ISO. Т.е. Стандарт С определяет тот набор функций, который должен поддерживать любой стандартный компилятор. Однако при этом большинство компиляторов предоставляют дополнительные функции, которые

не специфицированы в Стандарте. Например, многие компиляторы имеют функции работы с графикой, подпрограммы, управляемые с помощью мышки, и другие им подобные, которых нет в Стандарте С. Пока программа не переносится в другую вычислительную среду, эти нестандартные функции можно использовать без каких-либо негативных последствий. Но если программа должна быть переносимой, применение таких программ нужно ограничить.

Стандартная библиотека ANSI Си состоит из 24 заголовочных файлов, каждый из которых можно подключать к программному проекту при помощи одной директивы. Каждый заголовочный файл содержит объявления одной или более функций, определения типов данных и макросы.

- Файл с набором функций для работы с комплексными числами.
- Файл содержащий функции, используемые для классификации символов по их типам или для конвертации между верхним и нижним регистрами независимо от используемой кодировки
- Файл для проверки кодов ошибок, возвращаемых библиотечными функциями
- Файл для управления средой, использующей числа с плавающей точкой

8. Разбор командной строки.

Разбор аргументов командной строки

```
 #include <unistd.h>
intgetopt(int argc, char * const argv[], constchar *optstring);
 extern char *optarg;
 extern int opterr, optind, optopt;
```

Пример:

```
int c;
char *filename;
extern char *optarg;extern int optind, optopt, opterr;
while ((c = getopt(argc, argv, ":abf:")) != -1) {
 switch(c) {
 case 'a':   printf("a is set\n");                                    break;
case 'b':   printf("b is set\n");                                    break;
case 'f': filename = optarg;  printf("filename is %s\n", filename);  break;
case ':':    printf("-%c without filename\n", optopt);              break;
case '?':   printf("unknown arg %c\n", optopt);                     break;
} }
```

Linux: getopt_long(argc, argv, optstr, longopt, NULL);   //«Длинные опции»: --help

struct longopt { {“help”, 0, NULL, ‘h’}, … ,

{NULL, 0, NULL, 0}};

Функция getopt

int getopt(int argc, char * const argv[], const char * optstring)

int argc, char * const argv[] -параметры командной строки
const char * optstring - формат опций
int getopt - с каждым вызовом разбирается следующая короткая опция,возвращается символ опции (или ‘?’ - см. ниже);
если больше нечего разбирать, возвращает -1

Функция **getopt** считывает из глобальных переменных **<unistd.h>** положение места, где,предположительно, находится следующая опция, и пытается её разобрать и вернуть
В аргументе **optstring** описывается то, какие опции являются **допустимыми**,и у каких допустимых опций есть аргументы
Если опция разобрана и оказалась недопустимой, то возвращается символ ‘?’,а символ опции записывается в переменную optopt
Если разобрана опция с аргументом, то указатель на аргумент в массиве **argv** кладётся в переменную **optarg**

9. Разработка функций с переменным числом аргументов.

Функции с переменным числом параметров объявляются как обычные функции, но вместо недостающих аргументов ставится многоточие. Пусть мы хотим сделать функцию, которая складывает переданные ей числа, чисел может быть произвольное количество. Необходимо каким-то образом передать функции число параметров. Во-первых, можно явно передать число параметров обязательным аргументом. Во-вторых, последний аргумент может иметь некоторое «терминальное» значение, наткнувшись на которое функция закончит выполнение.

Функции с переменным числом аргументов

```
 #include <stdarg.h>
void va_start(va_list ap, argN);
void va_copy(va_list dest, va_list src);
type va_arg(va_list ap, type);
void va_end(va_list ap);
```

Возможная реализация:

```
#define va_start( ap, parm )        (ap)=(char*)(&(parm)+1)
#define va_arg( ap, type )          ((type*)((char*)(ap)+=sizeof(type)))[-1]
```

Применение:

```
int function_with-varargs(const char *args, ...){
va_list ap;
va_start(ap, args);
arg1 = va_arg(ap, int);
arg2 = va_arg(ap, double);
va_end(ap);
```

10. Функции работы с учетной информацией о пользователях.

```
#include <pwd.h>
struct passwd *getpwnam(constchar *);
struct passwd *getpwuid(uid_t);

void setpwent(void);
struct passwd *getpwent(void);
void endpwent(void);
struct passwd {
char    *pw_name;      //User's login name.
uid_t pw_uid;          //Numerical user ID.
gid_t pw_gid;          //Numerical group ID.
char    *pw_dir;       //Initial working directory.
char    *pw_shell;     //Program to use as shell.
};
```

11. Процессы. Атрибуты процесса.

Процесс –программа в стадии выполнения. Процесс = адресное пространство+ машинный код+ атрибуты.

Атрибуты: PID, PPID, UID, EUID, GID, … Класс, приоритет, значение nice, … Текущий и корневой каталог, переменные окружения, umask, … Параметры обработки сигналов, … Открытые файлы, …

Адресное пространство= TEXT + DATA (BSS + DYN) + STACK

- идентификатор процесса (PID);
- идентификатор родительского процесса (PPID);
- имя хозяина процесса;
- идентификатор хозяина процесса (UID);
- идентификатор группы хозяев (GUID);
- приоритет;
- терминал.

Идентификатор процесса — это просто номер его в порядке запуска, от нуля до максимального значения, возможного в данной системе (количество одновременно запущенных процессов — величина конечная). Минимальный (нулевой) номер обычно получает процесс `init`

Приоритет для каждого процесса устанавливается в тот момент, когда процесс порождается. Приоритет процесса определяется так называемым "значением nice", которое лежит в пределах от +20 (наименьший приоритет, процесс выполняется только тогда, когда ничто другое не занимает процессор), до -20 (наивысший приоритет).

Каждый процесс имеет маску **umask**, определяющую отключенные биты привилегий для создания файлов. Это позволяет процессу специфицировать достаточно либеральные права (обычно это касается общих прав на чтение и запись) и обеспечивать права, которые пользователь предпочитает. Если определенный файл особо важен, процесс создания может включать назначение более ограниченных прав, чем обычно, потому что umask никогда не влияет на менее строгие ограничения прав, а только на более строгие

12. Процессы: создание, завершение, ожидание.

Создание процесса

Новый процесс: fork() + exec().
fork – клонирование процесса
exec – замена существующего образом процесса из файла.

```
#include <unistd.h>
pid_t fork(void);
```

Пример:

```
pid_t p;
switch ( p = fork() ) {
case -1: ошибка…break;
case 0: код дочернего процесса; break;
default: printf(“PID дочернего= %d\n”, p ); }
```

13. Диспетчеризация процессов. Классы процессов.

Диспетчеризация процессов задач — это определение очерёдности получения процессора для процессов задач, находящихся в состоянии готовности, с целью их выполнения. Диспетчеризация процесса связана с его переводом из состояния готовности в состояние выполнения счёта. Диспетчеризация, для конкретного процесса, может выполняться многократно, т. е. процесс может несколько раз переходить из состояния готовности в состояние выполнения и обратно на интервале своего существования. Так как в каждый такт процессорного времени могут выполняться команды только одной задачи, диспетчеризация предполагает создание и модификацию очереди готовых к выполнению задач процессов. Элементами такой очереди как и других очередей в вычислительной системе на физическом уровне являются дескрипторы задач

Внутренняя синхронизация процессов основана на использовании аппарата событий для изменения состояний процессов в фазе «система». Состояние процесса отражает поле p_stat структуры struct proc дескриптора процесса. Процесс может находиться в одном из следующих состояний в соответствии со значением поля p_stat:
SONPROC - процесс выполняется, т.е. его инструкции обрабатываются процессором;
SRUN - процесс готов к выполнению, но не обладает приоритетом, достаточным для использования ресурсов процессора;
SSLEEP - процесс ожидает некоторое событие, до наступления которого он не претендует на ресурсы процессора;
SZOMB - состояние промежуточного завершения, когда процесс не имеет образа в RAM, но сохраняет дескриптор в таблице процессов.
В каждый момент времени только один процесс может являться текущим, т.е. использовать ресурсы процессора и находиться в состоянии SONPROC. Другие процессы, обработка которых не блокирована ожиданием событий, находятся в состоянии SRUN

https://lektsii.net/3-135764.html

14. Процессы реального времени.

Операционная система Linux обеспечивает две стратегии планирования в режиме реального времени (real-lime): `SCHED_FIFO` и `SCHED_RR`. Стратегия планирования `SCHED_OTHER` является обычной стратегией планирования, т.е. стратегий планирования не в режиме реального времени. Стратегия `SCHED_FIFO` обеспечивает простой алгоритм планирования по идеологии "первым вошел — первым обслужен" (first-in first-out, FIFO) без квантов времени. Готовое к выполнению задание со стратегией планирования `SCHED_FIFO` всегда будет планироваться на выполнение перед всеми заданиями со стратегией планирования `SCHED_OTHER`. Когда задание со стратегией `SCHED_FIFO` становится готовым к выполнению, то оно будет продолжать выполняться до тех пор, пока не заблокируется или пока явно не отдаст управление. Две или более задач с одинаковым приоритетом, имеющие стратегию планирования `SCHED_FIFO`, будут планироваться на выполнение по круговому алгоритму (round-robin). Если задание, имеющее стратегию планирования `SCHED_FIFO`, является готовым к выполнению, то все задачи с более низким приоритетом не могут выполняться до тех пор, пока это задание не завершится.

Стратегия `SCHED_RR` аналогична стратегии `SCHED_FIFO`, за исключением того, что процесс может выполняться только до тех пор, пока не израсходует предопределенный ему квант времени. Таким образом, стратегия `SCHED_RR` — это стратегия `SCHED_FIFO` с квантами времени, т.е. круговой алгоритм планирования (round-robin) реального времени. Когда истекает квант времени процесса со стратегией планирования SCHED_RR, то другие процессы с таким же приоритетом планируются по круговому алгоритму. Квант времени

используется только для того, чтобы перепланировать выполнение заданий с таким же приоритетом.

https://it.wikireading.ru/1760

15. Сигналы. Надежная и ненадёжная обработка сигналов.

Ненадежные сигналы - это те, для которых вызванный однажды обработчик сигнала не остается Такие "сигналы-выстрелы" должны перезапускать обработчик внутри самого обработчика, если есть желание сохранить сигнал действующим. Из-за этого возможна ситуация гонок, в которой сигнал может прийти до перезапуска обработчика - и тогда он будет потерян, или придет вовремя - и тогда сработает в соответствии с заданным поведением (например, убьет процесс). Такие сигналы ненадежны, поскольку отлов сигнала и переинсталяция обработчика не являются атомарными операциями.

В семантике ненадежных процессов системные вызовы не повторяются автоматически будучи прерванными поступившим сигналом. Поэтому для обеспечения отработки всех системных вызовов программа должна проверять значение errno после каждого из них и повторять вызовы, если это значение равно EINTR.

По тем же причинам семантика ненадежных сигналов не предоставляет легкого пути реализации атомарных пауз для усыпления процесса до получения сигнала. Ненадежное поведение постоянно перезапускающегося обработчика может привести к неготовности спящего в нужный момент принять сигнал.

Напротив, семантика надежных сигналов оставляет обработчик проинсталированным и ситуация гонок при перезапуске избегается. В то же время определенные сигналы могут быть запущены заново, а атомарная операция паузы доступна через функцию POSIX sigsuspend.

https://it.wikireading.ru/6556

16. Сигнальная маска.

sa_mask — маска сигналов который будут блокированы пока выполняется наш обработчик. + по дефолту блокируется и сам полученный сигнал

Добавление сигналов в структуру sigset_t sa_mask, ее очистка и т.п. осуществляются при помощи набора функций sigemptyset(), sigfillset(), sigaddset(), sigdelset(). Первые две функции принимают один параметр – указатель на структуру sigset_t. Эти функции очищают и заполняют всеми возможными сигналами структуру sigset_t соответственно.

Последние две функции, соответственно, добавляют и удаляют один определенный сигнал из структуры и имеют по два параметра. Их первый параметр – указатель на структуру sigset_t, а второй – номер сигнала.

Все рассмотренные выше функции возвращают 0 при успешном завершении и число, не равное нулю, – при ошибке.

Кроме того, существует еще одна функция, проверяющая, находится ли указанный сигнал в указанном наборе – sigismember(). Ее параметры совпадают с параметрами sigaddset(). Функция возвращает 1, если сигнал находится в наборе, 0 – если не находится, и отрицательное число – при возникшей ошибке.

Помимо всего прочего, мы можем задать список сигналов, доставка которых процессу будет заблокирована. Это выполняется при помощи функции sigprocmask(int how, const sigset_t * set, sigset_t * oldset).

Первый ее параметр описывает то, что должно выполняться:

1. SIG_BLOCK – сигналы из набора set блокируются;
2. SIG_UNBLOCK – сигналы из набора set разблокируются;
3. SIG_SETMASK – сигналы из набора set блокируются, остальные разблокируются.

Второй параметр является указателем на тот самый набор, сигналы из которого блокируются/разблокируются. Если он равен NULL, то значение первого параметра игнорируется системным вызовом.

Третий параметр – указатель на уже используемую маску сигналов; его можно поставить в NULL, если эти данные не нужны.

Для получения списка ожидающих сигналов можно использовать функцию sigpending(), которая принимает единственный параметр – указатель на структуру sigset_t, куда будет записан набор ожидающих сигналов.

17. Особенности обработки сигналов реального времени.

Сигналы могут быть отнесены к двум группам:

1. Сигналы реального времени, которые могут принимать значения между SIGRTMIN и SIGRTMAX включительно. Posix требует, чтобы предоставлялось по крайней мере RTSIG_MAX сигналов, и минимальное значение этой константы равно 8.

2. Все прочие сигналы: SIGALRM, SIGINT, SIGKILL и пр.

Отличия сигналов реального времени от «обычных» сигналов:
1. Приоритет.
2. Счетчик количества в очереди.
3. Доставка данных вместе с сигналом.

```
#include <signal.h>

int sigqueue( pid_t pid, int signo, const union sigval value );

union sigval{
int sival_int;     // Integer signal value.
void   *sival_ptr;    // Pointer signal value.
};
```

18. Управление памятью. Фиксирование страниц в памяти.

Система управления памятью в Linux осуществляет подкачку страниц по обращению в соответствии с COPY-ON-WRITE стратегией, основанной на механизме подкачки, который поддерживается 386-м процессором. Процесс получает свои таблицы страниц от родител я (при выполнении fork()) со входами, помеченными как READ-ONLY или замещаемые. Затем, если процесс пытается писать в эту область памяти, и страница является COPY-ON-WRITE страницей, она

копируется и помечается как READ-WRITE. Инструкция exec() приводит к считыванию страницы или то же самое происходит при выполнении программы.

Проблемы fork() и альтернативы

pid_t fork(void); создаются копиистраницпамятидля дочернегопроцесса

1) ПредложениеBSD pid_t vfork(void); страницы памятиродительскогопроцессасвязываются с дочерним

2) ПредложениеAT&T pid_t fork(void); страницы памятиродительскогопроцессапомечаются каксору-on-write исвязываютсяс дочерним.

Что меняется при fork()

PID, PPID, tms_utime, tms_stime, tms_cutime, and tms_cstime устанавливаются в 0.

Блокировки на файл, память, таймеры ожидаемые сигналы сбрасываются

таймеры, асинхронные операции ввода/вывода не наследуются.

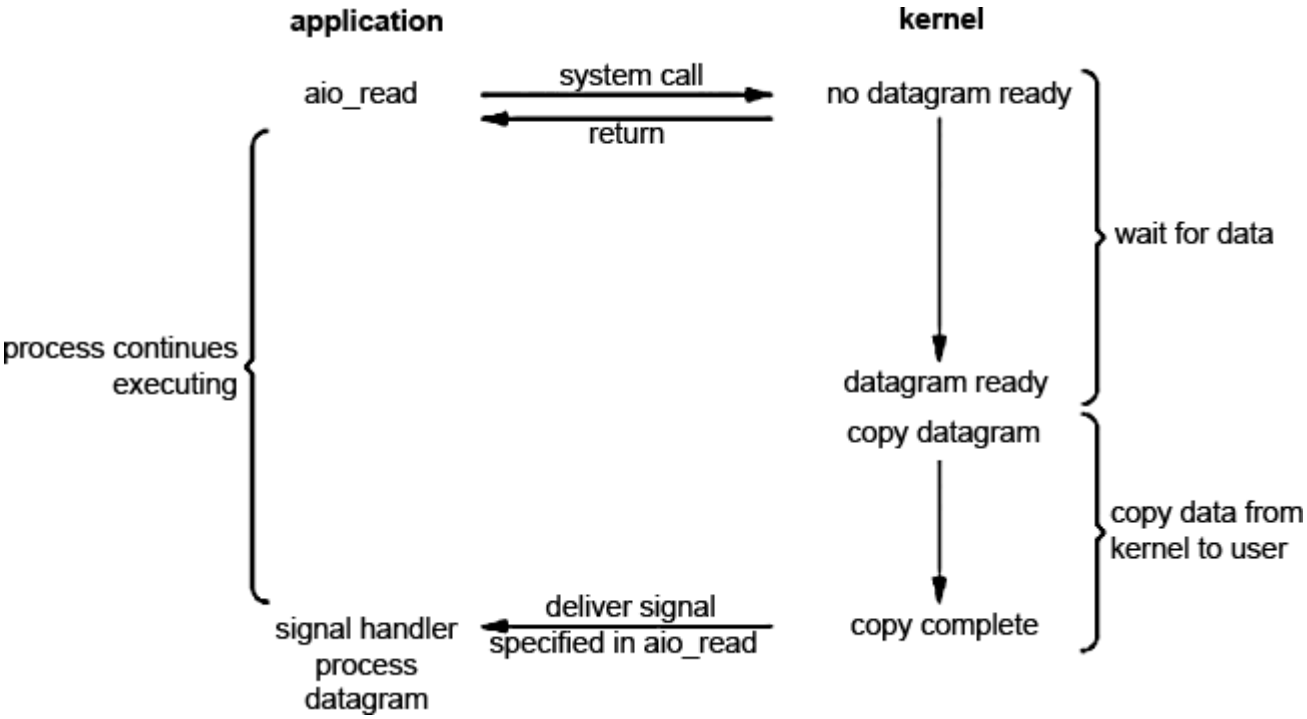
Счетчики времени CPU устанавливаются в 0.

Процесс создается с одним потоком.

Дескрипторы открытых файлов, текущий каталог копируются.

19. Асинхронный ввод/вывод.

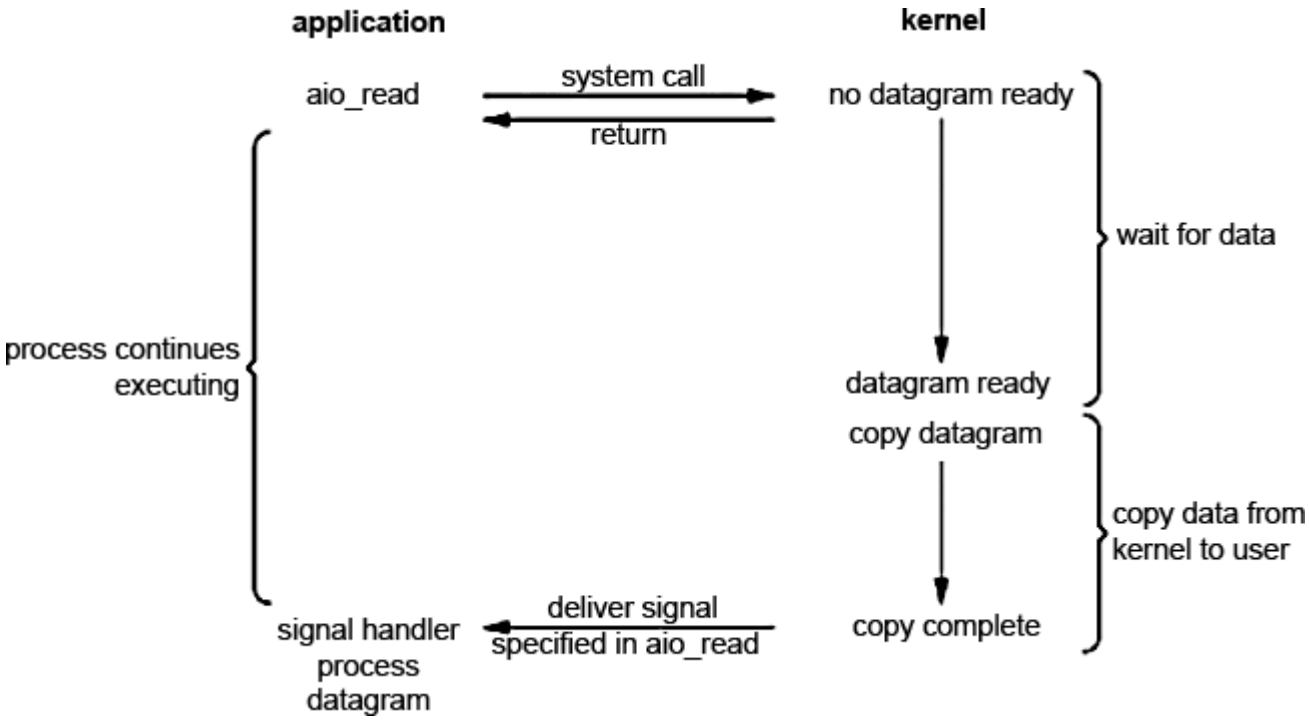
Асинхронный ввод/вывод осуществляется с помощью специальных системных вызовов. В основе лежит простая идея — ядру дается команда начать операцию и уведомить нас (с помощью сигналов, или еще как-то) когда операция ввода/вывода будет полностью завершена (включая копирование данных в буфер процесса). Это основное отличие данной реализации от реализации на сигналах. Схематично процессы асинхронного ввода вывода представлены на изображении.



Делаем системный вызов aio_read и указываем все необходимые параметры. Всю остальную работу делает за нас ядро. Конечно, должен существовать механизм который бы уведомил

процесс о том что I/O завершен.
асинхронный — значит независимый во времени. То есть единожды запущенный он живет своей жизнью пока не выполнится, а затем мы просто получаем результат

Асинхронный ввод/вывод осуществляется с помощью специальных системных вызовов. В основе лежит простая идея — ядру дается команда начать операцию и уведомить нас (с помощью сигналов, или еще как-то) когда операция ввода/вывода будет полностью завершена (включая копирование данных в буфер процесса). Это основное отличие данной реализации от реализации на сигналах. Схематично процессы асинхронного ввода вывода представлены на изображении.



Делаем системный вызов aio_read и указываем все необходимые параметры. Всю остальную работу делает за нас ядро. Конечно, должен существовать механизм который бы уведомил процесс о том что I/O завершен. И тут потенциально возникает множество проблем. Но об этом в другой раз.
Вообще с данным термином связано очень много проблем, примеры ссылок уже приводились. Часто происходит смешение понятий между асинхронным, неблокирующим и мультиплексированным вводом выводом, видимо потому что само понятие «асинхронный» может трактоваться по-разному. В моем понимании асинхронный — значит независимый во времени. То есть единожды запущенный он живет своей жизнью пока не выполнится, а затем мы просто получаем результат

https://habr.com/ru/post/111357/

20. Неименованные каналы.

Неименованный канал является средством взаимодействия между связанными процессами - родительским и дочерним. Родительский процесс создает канал при помощи системного вызова:

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

Херня какая-то https://life-prog.ru/view_linux.php?id=21

21. Именованные каналы.

Для передачи сообщений можно использовать механизмы сокетов, каналов, D-bus и другие технологии. Про сокеты на каждом углу можно почитать, а про D-bus отдельную статью написать. Поэтому я решил остановиться на малоозвученных технологиях отвечающих стандартам POSIX и привести рабочие примеры.

Рассмотрим передачу сообщений по именованным каналам. Схематично передача выглядит так:

Для создания именованных каналов будем использовать функцию, **mkfifo()**:

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

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

Функция создает специальный FIFO файл с именем **pathname**, а параметр **mode** задает права доступа к файлу.

Как только файл создан, любой процесс может открыть этот файл для чтения или записи также, как открывает обычный файл. Однако, для корректного использования файла, необходимо открыть его одновременно двумя процессами/потоками, одним для получение данных (чтение файла), другим на передачу (запись в файл).

https://habr.com/ru/post/122108/#part_namedpipe

22. Интервальные таймеры System V и BSD.

Интервальные таймеры, будучи активизированными, непрерывно передают сигналы в процесс на систематической основе. Точное значение термина *систематический* зависит от используемого интервального таймера. С каждым процессом ассоциированы три таймера.

`ITIMER_REAL`	Отслеживает время в терминах настенных часов — в реальном времени (в зависимости от выполнения процесса) — и генерирует `сигнал` SIGALRM. Несовместим с системным вызовом `alarm()`, который используется функцией `sleep()`. Не применяйте ни `alarm()`, ни `sleep()`, если имеется реальный интервальный таймер.

ITIMER_VIRTUAL	Подсчитывает время только при исполнении процесса — не учитывая системные вызовы, которые производит процесс — и генерирует сигнал SIGVTALRM.
ITIMER_PROF	Подсчитывает время только при выполнении процесса — включая время, за которое ядро посылает исполнительные системные вызовы от имени процесса, и не включая время, потраченное на прерывание процесса по инициативе самого процесса — и генерирует сигнал SIGPROF. Учет времени, затраченного на обработку прерываний, оказывается настолько трудоемким, что даже может изменить настройки таймера.

https://it.wikireading.ru/3440

23. Таймеры POSIX.

Вызов **timer_create**() создаёт новый таймер для процесса. Идентификатор нового таймера возвращается в буфере, указанном в *timerid*, его значение не должно быть равно null. Данный идентификатор уникален для процесса, пока таймер не будет удалён. Новый таймер создаётся неактивным.

В аргументе clockid задаются часы, которые используются в новом таймере для учёта времени. Это может быть одно из следующих значений:

- CLOCK_REALTIME

Настраиваемые системные часы реального времени.

- CLOCK_MONOTONIC

Ненастраиваемые, постоянно идущие вперёд часы, отсчитывающие время с некоторой неопределённой точки в прошлом, которая не изменяется с момент запуска системы.

- CLOCK_PROCESS_CPUTIME_ID (начиная с Linux 2.6.12)

Часы, измеряющие время ЦП (пользовательское и системное), затраченное вызывающим процессом (всеми его нитями).

- CLOCK_THREAD_CPUTIME_ID (начиная с Linux 2.6.12)

Часы, измеряющие время ЦП (пользовательское и системное), затраченное вызывающей нитью.

http://ru.manpages.org/timer_create/2

24. Очереди сообщений System V.

Каждой очереди сообщений System V сопоставляется свой *идентификатор очереди сообщений.* Любой процесс с соответствующими привилегиями может поместить сообщение в очередь, и любой процесс с другими соответствующими привилегиями может сообщение из очереди считать. Как и для очередей сообщений Posix, для помещения сообщения в очередь System V не требуется наличия подключенного к ней на считывание процесса.

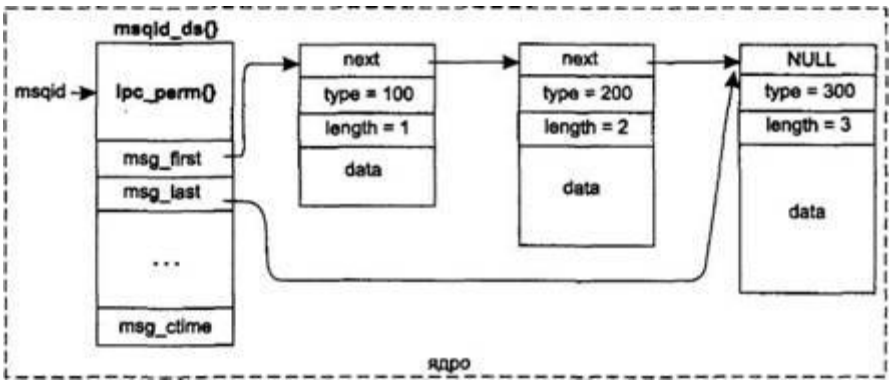


Рис. 6.1. Структура очереди system V в ядре

https://it.wikireading.ru/24815

25. Очереди сообщений POSIX.

Очередь сообщений можно рассматривать как связный список сообщений. Программные потоки с соответствующими разрешениями могут помещать сообщения в очередь, а потоки с другими соответствующими разрешениями могут извлекать их оттуда. Каждое сообщение представляет собой запись (вспомните сравнение потоков и сообщений в разделе 4.10), и каждому сообщению его отправителем присваивается приоритет. Для записи сообщения в очередь не требуется наличия ожидающего его процесса.

- операция считывания из очереди сообщений Posix всегда возвращает самое старое сообщение с наивысшим приоритетом, тогда как из очереди System V можно считать сообщение с произвольно указанным приоритетом;
- очереди сообщений Posix позволяют отправить сигнал или запустить программный поток при помещении сообщения в пустую очередь, тогда как для очередей System V ничего подобного не предусматривается.

Каждое сообщение в очереди состоит из следующих частей:

- приоритет (беззнаковое целое, Posix) либо тип сообщения (целое типа long, System V);
- длина полезной части сообщения, которая может быть нулевой;
- собственно данные (если длина сообщения отлична от 0).

https://it.wikireading.ru/24780

26. Семафоры System V.

Бинарный семафор

1. По своим свойствам такой семафор аналогичен взаимному исключению (глава 7), причем значение 0 для семафора соответствует блокированию ресурса, а 1 — освобождению.

семафор-счетчик, значение которого лежит в диапазоне от 0 до некоторого ограничения, которое, согласно Posix, не должно быть меньше 32767. Они использовался для подсчета доступных ресурсов в задаче о производителях и потребителях, причем значение семафора соответствовало количеству доступных ресурсов.

Для обоих типов семафоров операция *wait* состояла в ожидании изменения значения семафора с нулевого на ненулевое и последующем уменьшении этого значения на 1. Операция *post*

увеличивала значение семафора на 1, оповещая об этом все процессы, ожидавшие изменения значения семафора.

Для семафоров System V определен еще один уровень сложности:

набор семафоров-счетчиков — один или несколько семафоров, каждый из которых является счетчиком. На количество семафоров в наборе существует ограничение (обычно порядка 25). Когда мы говорим о семафоре System V, мы подразумеваем именно набор семафоров-счетчиков, а когда говорим о семафоре Posix, подразумевается ровно один семафор-счетчик.

https://it.wikireading.ru/24934

27. Семафоры POSIX.

Семафоры POSIX позволяют процессам и нитям синхронизировать свою работу.

Семафор представляет собой целое число, значение которого никогда не будет меньше нуля. Над семафорами выполняются две операции: увеличение значения семафора на единицу (**sem_post**(3)) и уменьшение значения семафора на единицу (**sem_wait**(3)). Если значение семафора равно нулю, то операция **sem_wait**(3) блокирует работу до тех пор, пока значение не станет больше нуля.

Есть два вида семафоров POSIX: именованные семафоры и безымянные семафоры.

Именованные семафоры

> Именованные семафоры отличают по именам вида */имя* — строка (с null в конце) до **NAME_MAX**-*4* (т. е., 251) символов, состоящая из начальной косой черты и одного или нескольких символов (символ косой черты не допускается). Два процесса могут работать с одним семафором указав его имя в **sem_open**(3).
>
> Функция **sem_open**(3) создаёт новый именованный семафор или открывает существующий. После открытия семафора с ним можно работать посредством **sem_post**(3) и **sem_wait**(3). Когда процесс закончил использовать семафор, его можно закрыть с помощью **sem_close**(3). Когда все процессы закончили использовать семафор, его можно удалить из системы с помощью **sem_unlink**(3).

Безымянные семафоры (семафоры в памяти)

> Безымянные семафоры не имеют имени. Семафор размещается в области памяти, которая доступна нескольким нитям (*общий семафор для нитей)* или процессам (*общий семафор для процессов*). Общий семафор для нитей размещается в области памяти, которая доступна из нитей процесса, например в глобальной переменной. Общий семафор для процессов должен размещаться в области общей памяти (например, в сегменте общей памяти System V, созданной с помощью **shmget**(2), или в объекте общей памяти POSIX, созданном с помощью **shm_open**(3)).
>
> Перед началом использования безымянный семафор должны быть проинициализирован с помощью **sem_init**(3). После этого с ним можно работать через **sem_post**(3) и **sem_wait**(3). Если семафор больше не нужен, то семафор нужно уничтожить с помощью **sem_destroy**(3) (но до освобождения выделенной для него памяти).

http://ru.manpages.org/sem_overview/7

28. Семафоры Sun.

29. Разделяемая память System V.

Основные принципы разделяемой памяти System V совпадают с концепцией разделяемой памяти Posix. Вместо вызовов shm_open и mmap в этой системе используются вызовы shmget и shmat.

Каналы и сокеты являются удобными средствами обмена информацией между процессами, но их использование при интенсивном обмене или обмене объемными данными приводит к значительным накладными расходам. Разделяемая память является специализированным средством взаимодействия, имеющим минимальные издержки использования.

В листинге ниже при помощи команды ipcs показаны созданные в системе сегменты разделяемой памяти, массивы семафоров и очереди сообщений — средства межпроцессного взаимодействия svipc, «унаследованные» Linux от W:[UNIX System V].

Экземпляры средств System VIPC идентифицируются при помощи глобально уникальных ключей, или локальных идентификаторов shmid (shared memory identifier), semid (semaphore identifier) и msqid (message queue identifier), и сродни файлам имеют владельцев и права доступа.

Идентификаторы IPC (сродни индексным и файловым дескрипторам файлов) изменяются при пересоздаиии/переоткрытии экземпляра, а ключи IPC (подобно именам файлов) — нет.

При помощи системного вызова **shmget** один из взаимодействующих процессов создает сегмент памяти, состоящий из страничных кадров без отображения на какой-либо файл. Впоследствии кадры этого сегмента при помощи системного вызова s**hmat** (shared memory attach) отображаются на страницы всех взаимодействующих процессов, за счет чего эти процессы и используют выделенную память совместно.

https://debianinstall.ru/razdelyaemaya-pamyat-semafory-i-ocheredi-soobshhenij-v-os-linux/

30. Отображение файлов в память.

файлы, отображаемые в память(memory-mapped files, далее — **MMF**).
Иногда их использование может дать довольно таки существенный прирост производительности по сравнению с обычной буферизированной работой с файлами.

Это механизм, который позволяет отображать файлы на участок памяти. Таким образом, при чтении данных из неё, производится считывание соответствующих байт из файла.

Когда мы обращаемся к памяти, в которую отображен файл, данные загружаются с диска в кэш(если их там ещё нет), затем делается отображение кэша в адресное пространство нашей программы. Если эти данные удаляются — отображение отменяется. Таким образом, мы избавляемся от операции копирования из кэша в буфер. Кроме того, нам не нужно париться по поводу оптимизации работы с диском — всю грязную работу берёт на себя ядро ОС.

Делается это с помощью функции **mmap()**.

Она возвращает адрес начала участка отображаемой памяти или **MAP_FAILED** в случае неудачи.
Первый аргумент — желаемый адрес начала участка отбраженной памяти. Не знаю, когда это может пригодится. Передаём 0 — тогда ядро само выберет этот адрес.
len — количество байт, которое нужно отобразить в память.
prot — число, определяющее степень защищённости отображенного участка памяти(только чтение, только запись, исполнение, область недоступна). Обычные значения — **PROT_READ**, **PROT_WRITE** (можно кобминировать через ИЛИ). Не буду на этом останавливаться — подробнее

читайте в манах. Отмечу лишь, что защищённость памяти не установится ниже, чем права, с которыми открыт файл.
flag — описывает атрибуты области. Обычное значение — **MAP_SHARED**. По поводу остальных — курите маны. Но замечу, что использование **MAP_FIXED** понижает переносимость приложения, т.к. его подержка является необязательной в POSIX-системах.
filedes — как вы уже догались — дескриптор файла, который нужно отобразить.
off — смещение отображенного участка от начала файла.

Подробнее https://habr.com/ru/post/55716/

31. Разделяемая память в стандарте POSIX.

тип межпроцессного взаимодействия — разделяемая память (*shared memory*). Схематично изобразим ее как некую именованную область в памяти, к которой обращаются одновременно два процесса:

Подробнее https://habr.com/ru/post/122108/#part_sharedmemory

Стандарт Posix.1 предоставляет два механизма совместного использования областей памяти для неродственных процессов:

1. Отображение файлов в память: файл открывается вызовом open, а его дескриптор используется при вызове mmap для отображения содержимого файла в адресное пространство процесса. Этот метод был описан в главе 12, и его использование было проиллюстрировано на примере родственных процессов. Однако он позволяет реализовать совместное использование памяти и для неродственных процессов.

2. Объекты разделяемой памяти: функция shm_open открывает объект IPC с именем стандарта Posix (например, полным именем объекта файловой системы), возвращая дескриптор, который может быть использован для отображения в адресное пространство процесса вызовом mmap. Данный метод будет описан в этой главе.

32. Файловая система: доступ, права доступа, атрибуты, жесткие и символические ссылки.

Изначально каждый файл имел **три параметра доступа**. Вот они:

- **Чтение** - разрешает получать содержимое файла, но на запись нет. Для каталога позволяет получить список файлов и каталогов, расположенных в нем;
- **Запись** - разрешает записывать новые данные в файл или изменять существующие, а также позволяет создавать и изменять файлы и каталоги;
- **Выполнение** - вы не можете выполнить программу, если у нее нет флага выполнения. Этот атрибут устанавливается для всех программ и скриптов, именно с помощью него система может понять, что этот файл нужно запускать как программу.

Подробнее **https://losst.ru/prava-dostupa-k-fajlam-v-linux**

Символические и жесткие ссылки - это особенность файловой системы Linux, которая позволяет размещать один и тот же файл в нескольких директориях. Это очень похоже на ярлыки в Windows,

так как файл на самом деле остается там же где и был, но вы можете на него сослаться из любого другого места.

Символические ссылки более всего похожи на обычные ярлыки. Они содержат адрес нужного файла в вашей файловой системе. Когда вы пытаетесь открыть такую ссылку, то открывается целевой файл или папка. Главное ее отличие от жестких ссылок в том, что при удалении целевого файла ссылка останется, но она будет указывать в никуда, поскольку файла на самом деле больше нет.

Вот основные особенности символических ссылок:

- Могут ссылаться на файлы и каталоги;
- После удаления, перемещения или переименования файла становятся недействительными;
- Права доступа и номер inode отличаются от исходного файла;
- При изменении прав доступа для исходного файла, права на ссылку останутся неизменными;
- Можно ссылаться на другие разделы диска;
- Содержат только имя файла, а не его содержимое.

Жесткий тип ссылок реализован на более низком уровне файловой системы. Файл размещен только в определенном месте жесткого диска. Но на это место могут ссылаться несколько ссылок из файловой системы. Каждая из ссылок - это отдельный файл, но ведут они к одному участку жесткого диска. Файл можно перемещать между каталогами, и все ссылки останутся рабочими, поскольку для них неважно имя. Рассмотрим особенности:

- Работают только в пределах одной файловой системы;
- Нельзя ссылаться на каталоги;
- Имеют ту же информацию inode и набор разрешений что и у исходного файла;
- Разрешения на ссылку изменяться при изменении разрешений файла;
- Можно перемещать и переименовывать и даже удалять файл без вреда ссылке.

https://losst.ru/simvolicheskie-i-zhestkie-ssylki-linux

33. Файловая система: работа с каталогами.

- **ls** - список файлов в директории;
- **cd** - переход между директориями;
- **rm** - удалить файл;
- **rmdir** - удалить папку;
- **mv** - переместить файл;
- **cp** - скопировать файл;
- **mkdir** - создать папку;
- **ln** - создать ссылку;
- **chmod** - изменить права файла;
- **touch** - создать пустой файл.

https://community.vscale.io/hc/ru/community/posts/211285805-%D0%A0%D0%B0%D0%B1%D0%BE%D1%82%D0%B0-%D1%81-%D1%84%D0%B0%D0%B9%D0%BB%D0%B0%D0%BC%D0%B8-%D0%B8-%D0%BA%D0%B0%D1%82%D0%B0%D0%BB%D0%BE%D0%B3%D0%B0%D0%BC%D0%B8-%D0%B2-Linux

34. Блокировки файлов.

Блокировка записей

Блокировка записи является блокировкой части файла. Поскольку файлы Unix являются просто потоками байтов, соответственно просто осуществляется блокировка диапазона байтов.

Блокировка всего файла

Блокировка всего файла, как предполагает название, блокирует весь файл, даже если его размер меняется в блокированном состоянии. Интерфейс BSD предусматривает блокирование лишь всего файла. Для блокирования всего файла с использованием интерфейса POSIX указывают нулевую длину. Это интерпретируется особым образом как «весь файл».

Функуции блокировки файлов: `flock()`,`fcntl()`, `lockf()`

35. Потоки. Диспетчеризация потоков.

поток - это выполнение последовательности машинных инструкций.

Диспетчеризация – выбор процессора и его времени для процесса, готового к выполнению.

Диспетчеризация заключается в реализации найденного в результате планиро-вания (динамического или статистического) решения, то есть в переключении процессора с одного потока на другой. Прежде чем прервать выполнение потока, ОС запоминает его контекст, с тем чтобы впоследствии использовать эту информацию для последующего возобновления выполнения данного потока.
Диспетчеризация сводится к следующему: сохранение контекста текущего потока, который требуется сменить; загрузка контекста нового потока, выбранного в результате планирования; запуск нового потока на выполнение.

Диспетчеризация имеет несколько дисциплин:

1) Бесприоритетные - линейные (в порядке очереди, случайный выбор процесса), циклические (циклический алгоритм, многоприоритетный циклический алгоритм)

2) Приоритетные – с динамическим приоритетом (приоритеты, зависящие от времени ожидания, зависящие от времени обслуживания), с фиксированным приоритетом (абсолютный, относительный приоритеты, адаптивное обслуживание, многоприоритетный циклический алгоритм).

36. Атрибуты потоков.

Атрибуты являются способом определить поведение потока, отличное от поведения по умолчанию. При создании потока с помощью `pthread_create`() или при инициализации переменной синхронизации может быть определен собственный объект атрибутов. Атрибуты определяются только во время создания потока; они не могут быть изменены в процессе использования.

Обычно вызываются три функции:

- инициализация атрибутов потока - `pthread_attr_init`() - создает объект `pthread_attr_t tattr` по умолчанию;
- изменение значений атрибутов (если значения по умолчанию не подходят) - разнообразные функции `pthread_attr_`*(), позволяющие установить значения индивидуальных атрибутов для структуры `pthread_attr_t tattr`;
- создание потока вызовом `pthread_create`() с соответствующими значениями атрибутов в структуре `pthread_attr_t tattr`.

37. Управление потоками Sun.

38. Управление потоками POSIX.

Стандарт POSIX специфицирует следующий набор функций для управления потоками:

- pthread_create(): создание потока
- pthread_exit():завершение потока (должна вызываться функцией потока при завершении)
- pthread_cancel():отмена потока
- pthread_join():заблокировать выполнение потока до прекращения другого потока, указанного в вызове функции
- pthread_detach():освободить ресурсы занимаемые потоком (если поток выполняется, то освобождение ресурсов произойдёт после его завершения)
- pthread_attr_init():инициализировать структуру атрибутов потока
- pthread_attr_setdetachstate():указать системе, что после завершения потока она может автоматически освободить ресурсы, занимаемые потоком
- pthread_attr_destroy():освободить память от структуры атрибутов потока (уничтожить дескриптор).

Имеются следующие примитивы синхронизации POSIX-потоков с помощью **мьютексов (mutexes)** – аналогов семафоров – и **условных переменных (conditional variables) –** оба эти типа объектов для синхронизации подробно рассмотрены позже в данном курсе:

- - pthread_mutex_init() – создание мьютекса;
- - pthread_mutex_destroy() – уничтожение мьютекса;
- - pthread_mutex_lock() – закрытие мьютекса;
- - pthread_mutex_trylock() – пробное закрытие мьютекса (если он уже закрыт, вызов игнорируется, и поток не блокируется);
- - pthread_mutex_unlock() – открытие мьютекса;
- - pthread_cond_init() – создание условной переменной;
- - pthread_cond_signal() – разблокировка условной переменной;
- - pthread_cond_wait() – ожидание по условной переменной.

https://www.intuit.ru/studies/courses/641/497/lecture/11284?page=3

39. Взаимоисключающие блокировки Sun.

40. Взаимоисключающие блокировки POSIX.

```
int pthread_mutex_init (pthread_mutex_t *mp, const pthread_mutex_attr_t
*mattrp)
```

инициализирует взаимоисключающую блокировку, выделяя необходимую память. Если *mattrp*=NULL, то создается блокировка с атрибутами "по умолчанию". В настоящее время атрибут один - область действия блокировки, его умолчательное значение - PTHREAD_PROCESS_PRIVATE (а может быть еще PTHREAD_PROCESS_SHARED).

int pthread_mutex_destroy (pthread_mutex_t *mp)
разрушает блокировку, освобождая выделенную память.

```
int pthread_mutex_lock (pthread_mutex_t *mp)
int pthread_mutex_unlock (pthread_mutex_t *mp)
int pthread_mutex_trylock (pthread_mutex_t *mp)
```

С помощью pthread_mutex_lock() поток пытается захватить блокировку. Если же блокировка уже принадлежит другому потоку, то вызывающий поток ставится в очередь (с учетом приоритетов потоков) к блокировке. После возврата из функции pthread_mutex_lock() блокировка будет принадлежать вызывающему потоку.

Функция pthread_mutex_unlock() освобождает захваченную ранее блокировку. Освободить блокировку может только ее владелец.

Функция pthread_mutex_trylock() - неблокирующая версия функции pthread_mutex_lock(). Если на момент обращения к этой функции блокировка уже захвачена, то происходит немедленный возврат из функции со значением EBUSY.

http://mintransmo.ru/Dev_bach/multithread.html

41. Условные переменные Sun.

42. Условные переменные POSIX.

Пример: Нам бы хотелось, чтобы поток ожидал события, не выполняя лишних действий, пока ему не придёт сообщение. После этого сообщения он начнёт работать. Например, поток будет ждать, пока не накопится 10 сообщений. После чего поток producer пошлёт ему сигнал. Consumer отработает и опять уйдёт ждать, пока не накопятся сообщения.

Для решения этой задачи можно использовать условную переменную. Условная переменная в Pthreads – это переменная типа pthread_cond_t, которая обеспечивает блокирование одного или нескольких потоков до тех пор, пока не придёт сигнал, или не пройдёт максимально установленное время ожидания. Условная переменная используется совместно с ассоциированным с ней мьютексом.

Как обычно, для создания условной переменной используется функция

*int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);*

где cond – указатель на переменную типа pthread_cond_t, attr – аттрибуты условной переменной.

https://learnc.info/c/pthreads_conditional_variables.html

43. Блокировки чтения/записи Sun.

44. Блокировки чтения/записи POSIX.

RW-lock (блокировка чтения-записи, блокировка «много читателей, один писатель») – это примитив синхронизации, который позволяет решить проблему «читателей-писателя» Проблема «много читателей, один писатель» довольно часто встречается при решении практических задач

многопоточного программирования. Такая проблема появляется тогда, когда много потоков читают данные, и один поток пишет (изменяет) данные. Простой подход – использованием мьютекса – считается медленным, так как нет необходимости блокировать ресурс, когда его только читают. Pthreads решает эту проблему с помощью блокировки чтения-записи, которая обеспечивает множественный доступ потоков читателей, и эксклюзивный доступ потока писателя.

В отличие от мьютекса, rwlock имеет два набора функций блокировки-освобождения ресурса – один для потоков читателей, другой для потока писателя. Инициализируется ресурс типа pthread_rwlock_t либо стандартным инициализатором PTHREAD_RWLOCK_INITIALIZER , либо функцией

*int pthread_rwlock_init(pthread_rwlock_t *rwlock, const pthread_rwlockattr_t *attr);*

которая принимает в качестве первого аргумента указатель на блокировку, в качестве второго – атрибуты вновь создаваемой блокировки.

Уничтожение блокировки производится с помощью функции

*int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);*

https://learnc.info/c/pthreads_rwlock.html

45. Данные потоков.

для структурирования драйверов символьных устройств и придания им модульности

решение реализовано в системе System V под названием потоков данных. Потоки данных основаны на возможности динамически соединять процесс пользователя с драйвером, а также динамически, во время исполнения, вставлять модули обра-ботки в поток данных. В некотором смысле поток представляет собой работающий в ядре аналог каналов в пространстве пользователя.

У потока данных всегда есть голова потока у вершины и соединение с драйве-ром у основания. В поток может быть вставлено столько модулей, сколько необ-ходимо. Обработка может происходить в обоих направлениях, так что каждому модулю может понадобиться одна секция для чтения (из драйвера) и одна секция для записи (в драйвер). Когда процесс пользователя пишет данные в поток, про-грамма в голове потока интерпретирует системный вызов и запаковывает данные в буферы потока, передаваемые от модуля к модулю вниз, при этом каждый мо-дуль выполняет соответствующие преобразования. У каждого модуля есть очередь чтения и очередь записи, так что буферы обрабатываются в правильном порядке. У модулей есть строго определенные интерфейсы, определяемые инфраструкту-рой потока, что позволяет объединять вместе несвязанные модули.

Важное свойство потоков данных – мультиплексирование. Мультиплексный модуль может взять один поток и расщепить его на несколько потоков или, наоборот, объединить несколько потоков в единый поток.

https://life-prog.ru/1_53566_potoki-dannih-v-UNIX.html

46. Двери.

Удаленный вызов процедуры (remote procedure call — RPC) происходит в ситуации, когда вызвавшая и вызываемая процедуры относятся к разным процессам. В такой ситуации мы обычно называем вызвавшую процедуру клиентом, а вызванную — сервером. Во втором сценарии на рис. 15.1 клиент и сервер выполняются на одном и том же узле. Это типичный частный случай третьего сценария, и это именно то, что осуществляется с помощью **дверей (doors**). Итак, двери дают возможность вызывать процедуру (функцию) другого процесса на том же узле. Один из процессов

(сервер) делает процедуру, находящуюся внутри него, доступной для вызова другим процессам (клиентам), создавая для этой процедуры дверь. Мы можем считать двери специальным типом IPC, поскольку при этом между процессами (клиентом и сервером) передается информация в форме аргументов функции и возвращаемых значений.

Чтобы воспользоваться интерфейсом дверей в Solaris 2.6, нужно подключить соответствующую библиотеку (-ldoor), содержащую функции door_XXX, описываемые в этой главе, и использовать файловую систему ядра (/kernel/sys/doorfs).

Внутри процесса двери идентифицируются дескрипторами. Извне двери могут идентифицироваться именами в файловой системе. Сервер создает дверь вызовом door_create; аргументом этой функции является указатель на процедуру, которая будет связана с данной дверью, а возвращаемое значение является дескриптором двери. Затем сервер связывает полное имя файла с дескриптором двери с помощью функции fattach. Клиент открывает дверь вызовом open, при этом аргументом функции является полное имя файла, которое сервер связал с дверью, а возвращаемым значением — дескриптор, который будет использоваться клиентом для доступа к двери. Затем клиент может вызывать процедуру с помощью door_call. Естественно, программа, являющаяся сервером для некоторой двери, может являться клиентом для другой.

Вызовы через двери являются синхронными: когда клиент вызывает door_call, возврата из этой функции не происходит до тех пор, пока процедура на сервере не завершит работу (возможно, с ошибкой)

https://it.wikireading.ru/24996

47. Вызов удаленных процедур (Sun RPC).

Программы, общающиеся через сеть, нуждаются в механизме связи. На нижнем уровне по поступлении пакетов подается сигнал, обрабатываемый сетевой программой обработки сигналов. На верхнем уровне работает механизм rendezvous (рандеву), принятый в языке Ада. В NFS используется механизм вызова удаленных процедур (RPC), в котором клиент взаимодействует с сервером (см. Рисунок 1). В соответствии с этим процессом клиент сначала обращается к процедуре, посылающей запрос на сервер. По прибытии пакета с запросом сервер вызывает процедуру его вскрытия, выполняет запрашиваемую услугу, посылает ответ, и управление возвращается клиенту.

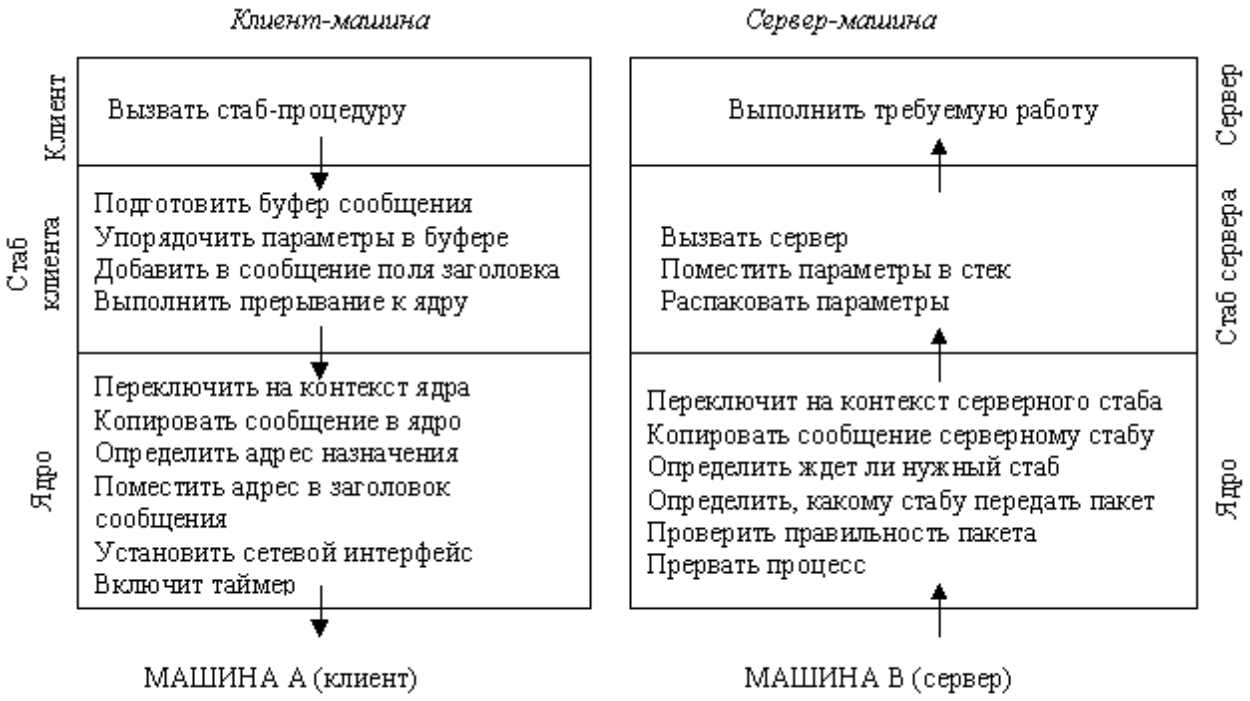


Рисунок 1. Сетевое взаимодействие через механизм RPC

Интерфейс RPC можно представить состоящим из трех уровней:

- Верхний уровень полностью "прозрачен". Программа этого уровня может, например, содержать обращение к процедуре rnusers(), возвращающей число пользователей на удаленной машине. Вам не нужно знать об использовании механизма RPC, поскольку вы делаете обращение в программе.

- Средний уровень предназначен для наиболее общих приложений. RPC-вызовами на этом уровне занимаются подпрограммы registerrpc() и callrpc(): registerrpc() получает общесис темный код, а callrpc() исполняет вызов удаленной процедуры. Вызов rnusers() реализуется с помощью этих двух подпрограмм.

- Нижний уровень используется для более сложных задач, изменяющих умолчания на значения параметров процедур. На этом уровне вы можете явно манипулировать гнездами, используемыми для передачи RPC-сообщений.

Как правило, вам следует пользоваться верхним уровнем и избегать использования нижних уровней без особой необходимости.

http://www.xserver.ru/computer/protokol/razn/6/

48. Интерфейс гнезд (сокеты). Модель клиент-сервер и одноранговая модель.

Существуют две основные архитектуры сети: одноранговая (peer-to-peer) и клиент/сервер (client/server), причем вторая практически вытеснила первую.

В **одноранговой сети** все компьютеры равны — имеют один ранг. Любой компьютер может выступать как в роли сервера, то есть предоставлять свои ресурсы (файлы, принтеры) другому компьютеру, так и в роли клиента, другими словами — использовать предоставленные ему ресурсы. Одноранговые сети преимущественно распространены в домашних сетях или небольших офисах. В самом простом случае для организации такой сети нужно всего лишь пара компьютеров, снабженных сетевыми платами и коаксиальный кабель. Компьютеры в

одноранговой сети объединяются в рабочие группы. Каждая рабочая группа имеет свой идентификатор — имя рабочей группы.

В **сети клиент/сервер** используется другой способ управления доступом — на уровне пользователей. В этом случае можно разрешить доступ к ресурсу только определенным пользователям. Например, ваш компьютер А через сеть могут использовать два пользователя: Иванов и Петров. К этому компьютеру подключен принтер, который можно использовать по сети. Но вы не хотите, чтобы кто угодно печатал на вашем принтере, и установили пароль для доступа к этому ресурсу. Если у вас одноранговая сеть, то любой, кто узнает этот пароль, сможет использовать ваш принтер. В случае с сетью клиент/сервер вы можете разрешить использовать ваш принтер только Иванову или только Петрову (можно и обоим).

После рассмотрения архитектуры одноранговой сети можно прийти к выводу, что единственное преимущество этой архитектуры — это ее простота и дешевизна. Сети клиент/сервер обеспечивают более высокий уровень производительности и безопасности.

В отличие от одноранговой сети, в сети клиент/сервер существует один или несколько главных компьютеров — серверов. Все остальные компьютеры сети называются клиентами или рабочими станциями (workstations). Как я уже писал выше, сервер — это специальный компьютер, который предоставляет определенные услуги другим компьютерам. Существуют различные виды серверов (в зависимости от предоставляемых ими услуг): серверы баз данных, файловые серверы, серверы печати (принт-серверы), почтовые серверы, Web-серверы и т.д.

https://it.wikireading.ru/12106

49. Интерфейс транспортного уровня TLI/XTI.

В области проектирования компьютерных сетей Интерфейс транспортного уровня (от англ. Transport Layer Interface) (TLI) был сетевым API, поддерживаемым AT&T UNIX System V Release 3 (SVR3) и Release 4 (SVR4). TLI был двойником (но в System V) программного интерфейса сокетов Беркли. TLI позднее был стандартизирован как XTI, то есть X/Open Transport Interface.

Первоначально ожидалось, что протоколы OSI вытеснят TCP/IP, и таким образом TLI разрабатывался, исходя из точки зрения, ориентированной на модель OSI, то есть по аналогии с транспортным уровнем OSI. Другими словами, TLI выглядит схоже с сокетами (с точки зрения API).

TLI и XTI никогда широко не использовались в качестве BSD-сокетов, и хотя они все еще поддерживаются в операционных системах, произошедших от SVR4, как например, Solaris (так же как и «классическая» Mac OS в виде Open Transport), именно сокеты приняты в качестве стандарта сетевых API.