16.12.2024

Объектные файлы. BLAS, LAPACK, MKL. История и архитектура Unix.

Филиппов Михаил Витальевич

m.filippov@g.nsu.ru

89232283872

Императивное программирование, 2024-2025

Новосибирский государственный университет

*НАСТОЯЩАЯ НАУКА

Давайте познакомимся

Филиппов Михаил Витальевич

- Окончил магистратуру ФФ НГУ
- Окончил аспирантуру ИТ СО РАН
- Являюсь м.н.с. ИТ СО РАН
- 7+ лет опыт в программировании C/C++

План презентации

Объектные файлы

35 минут

BLAS, LAPACK, MKL

20 минут

История и архитектура Unix

35 минут

План презентации

Объектные файлы

35 минут

BLAS, LAPACK, MKL

20 минут

История и архитектура Unix

35 минут

Введение

Мы рассмотрим 6 разделов

1. Двоичный интерфейс приложений. Здесь мы поговорим об ABI (application binary interface) и его значимости.
2. Форматы объектных файлов. Этот раздел посвящен разным форматам объектных файлов — как актуальным, так и устаревшим.
3. Переносимые объектные файлы. Здесь будут рассмотрены переносимые объектные файлы и самые первые продукты компиляции проекта на языке C.
4. Исполняемые объектные файлы. В этом разделе мы поговорим об исполняемых объектных файлах и объясним, как они создаются из разных переносимых объектных файлов.
5. Статические библиотеки. Этот раздел посвящен статическим библиотекам и их созданию.
6. Динамические библиотеки. Здесь речь пойдет о разделяемых объектных файлах. Будет продемонстрирован процесс их создания из разных переносимых объектных файлов и показан процесс их использования в программе.

Двоичный интерфейс приложений

Любые библиотеки и фреймворки, независимо от используемых в них технологий и языка написания, предоставляют определенный набор возможностей, известный как программный интерфейс приложения (Application Programming Interface, API). Если библиотеку нужно применять в другом коде, то это следует делать через ее API.

API — это публичный интерфейс библиотеки, которого должно быть достаточно для ее полноценного использования; все остальное считается «черным ящиком», внутрь которого нельзя заглянуть.

API — своего рода соглашение, принятое двумя программными компонентами, которые обслуживают или используют друг друга. Концепция ABI имеет похожее назначение, только на другом уровне. Если API обеспечивает совместимость двух программных компонентов с точки зрения их функционального взаимодействия, то ABI гарантирует, что две программы и их соответствующие объектные файлы совместимы на уровне машинных инструкций.

ABI обычно содержит следующую информацию:

- набор инструкций целевой архитектуры, включая инструкции процессора, структуры памяти, порядок следования байтов, регистры и т. д.;
- существующие типы данных, их размеры и правила выравнивания;
- соглашение о вызове функций, включая такие подробности, как структура стекового фрейма и порядок размещения аргументов в стеке;
- механизм системных вызовов в Unix-подобных ОС;
- используемые форматы объектных файлов, включая переносимые, исполняемые и разделяемые;
- если речь идет об объектных файлах, сгенерированных компилятором C++, то в состав ABI также входит метод декорирования имен и структура виртуальной таблицы.

Форматы объектных файлов

В Unix-подобных операционных системах, таких как Linux и BSD, самым распространенным стандартом ABI является System V. В рамках System V ABI по умолчанию используется формат объектных файлов ELF **(Executable and Linking Format — формат выполнения и компоновки).**

На каждой платформе для хранения машинных инструкций используется свой формат объектных файлов. Обратите внимание: речь здесь идет о структуре самих файлов, а не о наборе инструкций, которые поддерживает архитектура.

Форматы, используемые в разных операционных системах.

- ELF применяется в Linux и многих других Unix-подобных ОС.
- Mach-O используется в системах OS X (macOS и iOS).
- PE применяется в Microsoft Windows.

Если обратиться к истории, то можно сказать, что все существующие на сегодняшний день форматы объектных файлов происходят от старого формата a.out, который был разработан для ранних версий Unix.

Название расшифровывается как assembler output (вывод ассемблера).

Но вскоре на смену a.out пришел другой формат, COFF (Common Object File Format — общий формат объектных файлов), ставший впоследствии основой для. В рамках выпуска OX/X компания Apple создала свою замену a.out под названием Mach-O. В Windows в качестве формата объектных файлов используется PE (Portable Execution — переносимый исполняемый файл), основанный на COFF.

Более подробную информацию о формате ELF и его структуре можно найти по адресу https://www.uclibc.org/docs/psABI-x86_64.pdf. Обратите внимание: в этом документе идет речь о System V ABI для 64-битной архитектуры AMD (amd64).

Переносимые объектные файлы

Переносимые объектные файлы создаются на этапе компиляции в машинный код в ходе сборки проекта. Они считаются промежуточными и используются в качестве ингредиентов для создания дальнейших и конечных продуктов. В связи с этим мы уделим им дополнительное внимание и изучим их содержимое.

В переносимом объектном файле, полученном из скомпилированной единицы трансляции, можно найти следующие элементы:

- инструкции машинного уровня, сгенерированные из функций, найденных в единице трансляции (код);
- значения инициализированных глобальных переменных, объявленных в единице трансляции (данные);
- таблицу символов, содержащую все символы, которые были объявлены и на которые ссылается единица трансляции.

Переносимые объектные файлы связаны с процессом, в ходе которого компоновщик формирует более крупный объектный файл — исполняемый или разделяемый.

funcs.c

```c
int max(int a, int b) {
    return a > b ? a : b;
}
int max_3(int a, int b, int c) {
    int temp = max(a, b);
    return c > temp ? c : temp;
}
```

main.c

```c
int max(int, int);
int max_3(int, int, int);
int a = 5, b = 10;
int main(int argc, char **argv){
    int m1 = max(a, b);
    int m2 = max_3(5, 8, -1);
    return 0;
}
```

Переносимые объектные файлы

```
$ gcc -c funcs.c -o funcs.o
$ gcc -c main.c -o main.o
$ readelf -hSl funcs.o
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Advanced Micro Devices X86-64
  ……
  Number of section headers:         13
  Section header string table index: 12

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS         0000000000000000  00000040
       000000000000004f  0000000000000000  AX       0     0     1
…
  [ 3] .data             PROGBITS         0000000000000000  0000008f
       0000000000000000  0000000000000000  WA       0     0     1
  [ 4] .bss              NOBITS           0000000000000000  0000008f
       0000000000000000  0000000000000000  WA       0     0     1
…
  [10] .symtab           SYMTAB           0000000000000000  00000138
       0000000000000078  0000000000000018          11     3     8
…
```

Содержимое объектного файла funcs.o в формате ELF

Секция **.text** содержит все машинные инструкции для единицы трансляции. В секциях **.data и .bss** находятся соответственно значения для инициализированных глобальных переменных и количество байтов, необходимых для неинициализированных глобальных переменных. Секция **.symtab** хранит таблицу символов.

Переносимые объектные файлы

Таблица символов объектного файла funcs.o

```
$readelf -s funcs.o
Symbol table '.symtab' contains 5 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS funcs.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000000    27 FUNC    GLOBAL DEFAULT    1 max
     4: 000000000000001b    52 FUNC    GLOBAL DEFAULT    1 max_3
```

Таблица символов объектного файла main.o

```
$ readelf -s main.o
Symbol table '.symtab' contains 8 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS main.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 0000000000000000     4 OBJECT  GLOBAL DEFAULT    3 a
     4: 0000000000000004     4 OBJECT  GLOBAL DEFAULT    3 b
     5: 0000000000000000    73 FUNC    GLOBAL DEFAULT    1 main
     6: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND max
     7: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND max_3
```

Символы, относящиеся к глобальным переменным a и b, а также символ функции main размещены по адресам, которые не похожи на итоговые. Это признак переносимого объектного файла.

```
$ gcc funcs.o main.o -o a.out
$ readelf -hSl a.out
ELF Header:
  ...
  Type:                              DYN (Position-Independent Executable file)
  Machine:                           Advanced Micro Devices X86-64
  ...
Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000000318  00000318
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.gnu.pr[...] NOTE             0000000000000338  00000338
       0000000000000030  0000000000000000   A       0     0     8
...
  [28] .shstrtab         STRTAB           0000000000000000  000035e5
       000000000000010c  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), l (large), p (processor specific)
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000000318 0x0000000000000318
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  ...
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000002df0 0x0000000000003df0 0x0000000000003df0
                 0x0000000000000210 0x0000000000000210  R      0x1
 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
...
   12     .init_array .fini_array .dynamic .got
```

Исполняемые объектные файлы – это один из конечных продуктов компиляции проекта на языке C. Они содержат те же элементы, что и их переносимые аналоги: машинные инструкции, значения для инициализированных глобальных переменных и таблицу символов. Отличается только расположение этих элементов.

Исполняемые объектные файлы

Вывод по ELF предыдущего слайда:

- С точки зрения формата ELF этот объектный файл является разделяемым. Иными словами, в ELF исполняемый объектный файл отличается от разделяемого только наличием определенных сегментов наподобие INTERP. Этот сегмент (который на самом деле ссылается на секцию .interp) используется загрузчиком для запуска и выполнения исполняемого файла.
- Четыре сегмента выделены жирным шрифтом. Первый, INTERP, уже рассмотрен в предыдущем пункте. Второй, TEXT, содержит все секции с машинными инструкциями. В третьем, DATA, находятся все значения, которые будут использоваться для инициализации глобальных переменных и других ранних структур. Четвертый сегмент ссылается на секцию с информацией, которая относится к динамической компоновке, такой как местоположение разделяемых объектных файлов, загружаемых во время выполнения.
- Если сравнивать с переносимым разделяемым объектным файлом, здесь есть больше секций, которые, вероятно, содержат данные, необходимые для загрузки и выполнения программы.

Исполняемые объектные файлы

```
$ readelf -s a.out

Symbol table '.dynsym' contains 6 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND _[...]@GLIBC_2.34 (2)
     2: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregiste
     3: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     4: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_register
     5: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND [...]@GLIBC_2

Symbol table '.symtab' contains 40 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS Scrt1.o
     2: 000000000000038c    32 OBJECT  LOCAL  DEFAULT    4 __abi_tag
    ...
    21: 0000000000004000     0 NOTYPE  WEAK   DEFAULT   23 data_start
    22: 0000000000001144    52 FUNC    GLOBAL DEFAULT   14 max_3
    23: 0000000000004014     4 OBJECT  GLOBAL DEFAULT   23 b
    24: 0000000000004018     0 NOTYPE  GLOBAL DEFAULT   23 _edata
    25: 00000000000011c4     0 FUNC    GLOBAL HIDDEN    15 _fini
    26: 0000000000001129    27 FUNC    GLOBAL DEFAULT   14 max
    ...
    38: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@G[...]
    39: 0000000000001000     0 FUNC    GLOBAL HIDDEN    11 _init
```

Исполняемый объектный файл содержит две разные таблицы символов. Первая, .dynsym, включает символы, которые должны быть разрешены во время загрузки программы, а во второй, .symtab, находится все то же, что и в динамической таблице, плюс все разрешенные символы.
В исполняемом объектном файле абсолютными являются обычные адреса, а в разделяемом — относительные.

Статические библиотеки

Статические библиотеки — один из возможных продуктов компиляции проекта на языке C.

Статическая библиотека в Unix — обычный архив с переносимыми объектными файлами. Как правило, она компонуется с другими объектными файлами в целях получения итоговой исполняемой программы.

Обратите внимание: сами по себе статические библиотеки не являются объектными файлами. Они скорее служат их контейнерами. Иными словами, это не ELFфайлы в Linux и не Mach-O-файлы в macOS. Это просто архивы, созданные Unix-утилитой ar.

Перед компоновкой из статической библиотеки извлекаются переносимые объектные файлы. Затем компоновщик ищет в них неопределенные символы и пытается их разрешить.

В системах семейства Unix применительно к статическим библиотекам действует общепринятое соглашение об именовании. Имя файла должно начинаться с lib и иметь расширение .a. В разных операционных системах могут действовать разные правила; например, в Microsoft Windows статические библиотеки имеют расширение .lib. Примерно так выглядит создание статической библиотеки из множества переносимых объектных файлов:

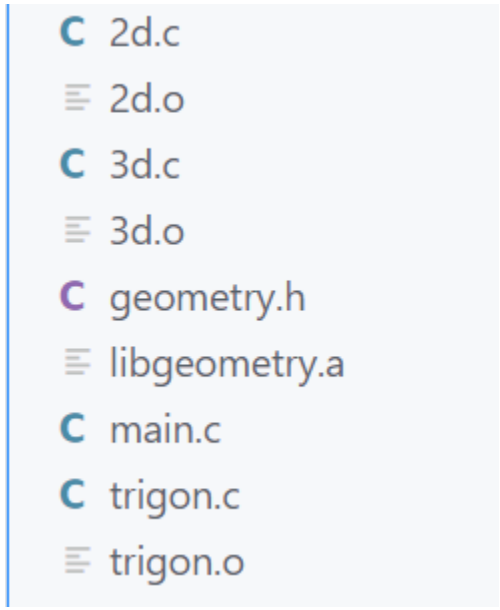
```
$ ar crs libexample.a aa.o bb.o ... zz.o
```

Про аргумент crs

Статические библиотеки

Библиотека для вычислительной геометрии пример с ноутбука.
Будет доступен по ссылке где и остальной (22_1)

```
$ gcc -c trigon.c -o trigon.o
$ gcc -c 2d.c -o 2d.o
$ gcc -c 3d.c -o 3d.o
$ ar crs libgeometry.a trigon.o 2d.o 3d.o
$ ar t libgeometry.a
trigon.o
2d.o
3d.o
```



Статические библиотеки

main.c

```c
#include <stdio.h>
#include "geometry.h"
int main(int argc, char **argv)
{
    cartesian_pos_2d_t cartesian_pos;
    cartesian_pos.x = 100;
    cartesian_pos.y = 200;
    polar_pos_2d_t polar_pos =
        convert_to_2d_polar_pos(&cartesian_pos);
    printf("Polar Position: Length: %f, Theta: %f (deg)\n",
           polar_pos.length, polar_pos.theta);
    return 0;
}
```

```
$ gcc -c main.c -o main.o
$ gcc main.o -
L/mnt/c/Users/Mfili/Desktop/learningc++/imperative_programming/lecti
ons_PowerPoint/lection_22_1 -lgeometry -lm -o a.out
$ ./a.out
Polar Position: Length: 223.606798, Theta: 63.434949 (deg)
```

Статические библиотеки

- Параметр -L/../lesson22_1 сообщает компилятору gcc, что каталог /../lesson22_1 входит в число тех мест, в которых можно найти статические и разделяемые библиотеки. По умолчанию компоновщик ищет библиотечные файлы в традиционных каталогах, таких как /usr/lib или /usr/local/lib. Если не указать параметр -L, то компоновщик выполнит поиск только по этим стандартным путям.
- Параметр -lgeometry говорит компилятору gcc, что ему нужно искать файл libgeometry.a или libgeometry.so. Расширение принадлежит разделяемым библиотекам, о которых мы поговорим в следующем разделе.
- Параметр -lm сообщает компилятору gcc, что нужно искать еще одну библиотеку с именем libm.a или libm.so. Она хранит определения математических функций, доступных в glibc, — в частности, cos, sin и acos. Обратите внимание: мы собираем пример 3.3 на компьютере под управлением Linux, поэтому в качестве реализации стандартной библиотеки C используется glibc. В macOS и, возможно, в других операционных системах данный параметр можно опустить.
- Параметр -o a.out говорит компилятору gcc, что итоговый исполняемый файл должен называться a.out.

Стоит отметить, что после компоновки программа не будет зависеть от наличия статических библиотек, поскольку все их содержимое встраивается в ее исполняемый файл. Иными словами, полученная программа является самостоятельной и для ее запуска не требуется присутствие статической библиотеки.

Динамические библиотеки

Динамические (они же разделяемые) библиотеки — еще один продукт компиляции с возможностью повторного использования. От статических библиотек они отличаются тем, что не входят в состав итогового исполняемого файла. Вместе этого их необходимо загружать и подключать во время запуска процесса.

В динамических библиотеках, неопределенные символы могут оставаться и после работы компоновщика; их поиск начинается в момент, когда программа готовится к загрузке и выполнению. Если у нас есть неопределенные динамические символы, то этап компоновки нужно видоизменить. Во время загрузки исполняемого файла и его подготовки к выполнению в виде процесса используется динамический компоновщик/загрузчик.

Неопределенные динамические символы должны загружаться из разделяемых объектных файлов — близких родственников статических библиотек. В большинстве Unix-подобных систем разделяемые объектные файлы имеют расширение .so, а в macOS — .dylib.

Перед самим запуском процесса разделяемый объектный файл загружается в участок памяти, доступный этому процессу. Данная процедура выполняется динамическим компоновщиком (или загрузчиком), который загружает и выполняет программу.

Во-первых, символы имеют относительные и абсолютные адреса, что позволяет загружать их в рамках сразу нескольких процессов.

Во-вторых, в разделяемых объектных файлах, в отличие от исполняемых, нет сегментов для загрузки программы. Это фактически означает, что разделяемые объектные файлы нельзя выполнять.

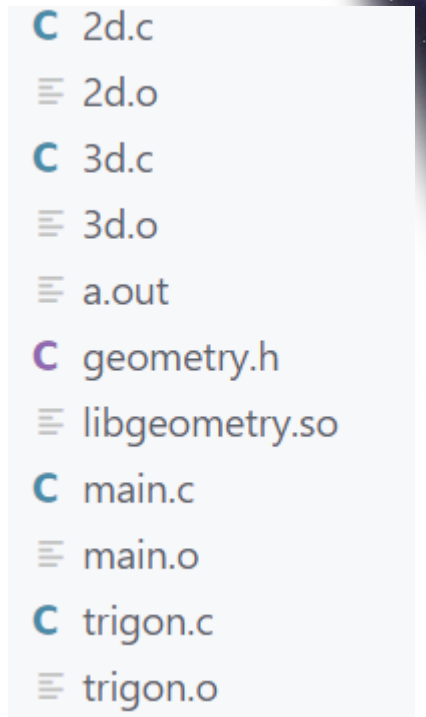
Динамические библиотеки

Дополнительный параметр **-fPIC** обязателен, если вам нужно создать динамическую библиотеку из набора переносимых объектных файлов. **PIC** расшифровывается как position independent code (позиционно независимый код).

Более подробную информацию о том, как работает динамическая загрузка программ. и разделяемых объектных файлов.

Чтобы создать динамическую библиотеку, нам снова нужно будет воспользоваться компилятором (в нашем случае это gcc). В отличие от статической библиотеки, которая является обычным архивом, разделяемый объектный файл остается объектным.

```
$ gcc -c trigon.c -fPIC -o trigon.o
$ gcc -c 2d.c -fPIC -o 2d.o
$ gcc -c 3d.c -fPIC -o 3d.o
$ gcc main.o -
L/mnt/c/Users/Mfili/Desktop/learningc++/imperative_programming/lections_Po
werPoint/lection_22_1 -lgeometry -lm -o a.out
$ ./a.out
./a.out: error while loading shared libraries: libgeometry.so: cannot open
shared object file: No such file or directory
$ export
LD_LIBRARY_PATH=/mnt/c/Users/Mfili/Desktop/learningc++/imperative_programm
ing/lections_PowerPoint/lection_22_1
$ ./a.out
Polar Position: Length: 223.606798, Theta: 63.434949 (deg)
```



Ручная загрузка разделяемых библиотек

```c
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
#include "geometry.h"
polar_pos_2d_t (*func_ptr)(cartesian_pos_2d_t *);
int main(int argc, char **argv) {
    void *handle =
dlopen("/mnt/c/Users/Mfili/Desktop/learningc++/imperative_programming/lect
ions_PowerPoint/lection_22_/libgeometry.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
    func_ptr = dlsym(handle, "convert_to_2d_polar_pos");
    if (!func_ptr) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
    cartesian_pos_2d_t cartesian_pos;
    cartesian_pos.x = 100;
    cartesian_pos.y = 200;
    polar_pos_2d_t polar_pos = func_ptr(&cartesian_pos);
    printf("Polar Position: Length: %f, Theta: %f (deg)\n",
           polar_pos.length, polar_pos.theta);
    return 0;
}
```

Разделяемые объектные файлы можно загружать и использовать иначе, не задействуя загрузчик (динамический компоновщик), который делает это автоматически. Идея в том, что программист загружает динамическую библиотеку вручную непосредственно перед использованием ее символов (функций).

Ручная загрузка разделяемых библиотек

Параметр **-lm** который позволит выполнить компоновку разделяемого объектного файла со стандартной математической библиотекой libm.so. Мы используем его в связи с тем, что в ходе ручной загрузки файла libgeometry.so его зависимости должны загружаться автоматически. В противном случае мы получим сообщения об отсутствии таких символов, как cos или sqrt, которые нужны самой библиотеке libgeometry.so. Обратите внимание: стандартная математическая библиотека не компонуется с итоговым исполняемым файлом; она будет разрешена автоматически во время загрузки libgeometry.so.

```
$ gcc -shared 2d.o 3d.o trigon.o -lm -o libgeometry.so
$ gcc main_2.c -ldl -o a_2.out
$ ./a_2.out
Polar Position: Length: 223.606798, Theta: 63.434949 (deg)
```

План презентации

Объектные файлы	BLAS, LAPACK, MKL	История и архитектура Unix
35 минут	20 минут	35 минут

Общее состояние библиотек BLAS, LAPACK и MKL

Библиотеки линейной алгебры: BLAS, LAPACK, MKL

- ***BLAS (Basic Linear Algebra Subprograms):*** Базовые операции с векторами и матрицами. Оптимизированы по скорости, используются как основа для других библиотек.
- ***LAPACK (Linear Algebra Package):*** Расширяет BLAS, предоставляя алгоритмы для решения систем уравнений, собственных значений и разложений матриц.
- ***MKL (Intel Math Kernel Library):*** Коммерческая реализация BLAS и LAPACK от Intel, оптимизирована для процессоров Intel, включает дополнительные функции (например, FFT).
- Состояние:
  - BLAS и LAPACK — стандарты с открытыми референсными реализациями (Netlib), но для высокой производительности используют оптимизированные версии (OpenBLAS, MKL).
  - MKL — лидер по производительности на Intel CPU, бесплатен для личного использования с 2021 года.
  - Активно применяются в научных вычислениях, машинном обучении и симуляциях.
- Зачем в C?: Интерфейсы на Fortran, но легко интегрируются в C через заголовки (cblas.h) и компиляцию.

Что такое BLAS?

- BLAS — это набор подпрограмм для базовых операций линейной алгебры:
  - Уровень 1: операции с векторами (скалярное произведение, норма).
  - Уровень 2: матрично-векторные операции (умножение матрицы на вектор).
  - Уровень 3: матрично-матричные операции (умножение матриц).
- Реализация:
  - Референсная версия: http://www.netlib.org/blas/
  - Оптимизированная: OpenBLAS (https://github.com/OpenMathLib/OpenBLAS).
- Зачем?: Высокая производительность благодаря оптимизации под архитектуру процессора.

Установка BLAS (OpenBLAS)

OpenBLAS — популярная открытая реализация BLAS.

- Установка на Ubuntu:

```
sudo apt update
sudo apt install libopenblas-dev
```

- • Компиляция из исходников:

1. Скачать с GitHub:

```
git clone https://github.com/OpenMathLib/OpenBLAS.git
```

2. Сборка:
3. cd OpenBLAS
4. make

```
sudo make install
```

5. Библиотека обычно устанавливается в /usr/local/lib.

- Проверка: После установки доступны файлы libopenblas.a и libopenblas.so.
- Связывание с C:

```
gcc -o program program.c -I/usr/local/include -L/usr/local/lib -lopenblas
```

Пример кода с BLAS (уровень 1)

Задача: Вычислить скалярное произведение двух векторов с помощью cblas_ddot.

```c
#include <stdio.h>
#include <cblas.h>
int main()
{
    int n = 3;
    double x[] = {1.0, 2.0, 3.0};
    double y[] = {4.0, 5.0, 6.0};
    double result;
    // Скалярное произведение: result = x^T * y
    result = cblas_ddot(n, x, 1, y, 1);
    printf("Скалярное произведение: %.2f\n", result); // Ожидаем 32.0
    return 0;
}
```

```
$ gcc main.c -o main –lopenblas
$ ./main
Скалярное произведение: 32.00
```

Пример кода с BLAS (уровень 3)

Задача: Умножить две матрицы A (2x3) и B (3x2) с помощью cblas_dgemm.

```c
#include <stdio.h>
#include <cblas.h>
int main()
{
    double A[] = {1, 2, 3, 4, 5, 6};    // 2x3
    double B[] = {7, 8, 9, 10, 11, 12}; // 3x2
    double C[4] = {0};                  // 2x2
    double alpha = 1.0, beta = 0.0;
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                2, 2, 3, alpha, A, 3, B, 2, beta, C, 2);
    printf("C = [%f, %f; %f, %f]\n", C[0], C[1], C[2], C[3]);
    return 0;
}
```

```
$ gcc main.c -o main -lopenblas
$ ./main
C = [58.000000, 64.000000; 139.000000, 154.000000]
```

История BLAS

Происхождение:

Разработан в 1970-х годах (BLAS Level 1 — 1979, авторы: Чарльз Лоусон и др.).

Цель: стандартизировать базовые операции для научных вычислений на Fortran.

Эволюция:

Level 2 (1988): матрично-векторные операции.

Level 3 (1990): матрично-матричные операции для суперкомпьютеров.

Интересный факт:

BLAS изначально был написан на Fortran, но его идеи настолько универсальны, что он стал основой для GPU-библиотек вроде cuBLAS (NVIDIA).

Современность:

Используется в машинном обучении (TensorFlow, PyTorch) через оптимизированные реализации (MKL, OpenBLAS).

Актуальность:

BLAS остаётся актуальной, потому что её идеи универсальны: она решает базовые задачи линейной алгебры, которые лежат в основе почти всех вычислений — от физики до машинного обучения. Её простота и стандартизация позволяют легко адаптировать код под новые архитектуры, а оптимизированные реализации, такие как OpenBLAS или MKL, обеспечивают высокую производительность даже на современных процессорах и GPU. К тому же, BLAS изначально проектировался с учётом переносимости — код на Fortran работал на любых машинах 1970-х, а сегодня его просто "ускоряют" под новые технологии."

Еще о BLAS

Почему BLAS быстр?
Использует кэш-память процессора эффективно.
Пример: *умножение матриц (Level 3) разбивается на блоки, Блочная версия минимизирует обращения к памяти.*
Реальный пример:
Референсный BLAS: ~10 MFLOPS (миллионов операций в секунду).
OpenBLAS на современном CPU: до 100 GFLOPS (гигафлопс).
Трюк оптимизации: Векторные инструкции (SIMD): AVX, SSE в OpenBLAS/MKL позволяют обрабатывать 4–8 чисел за один цикл.
Тест производительности (22_1 пример):
$ ./main
Чистый C: 8.300392 сек
BLAS: 0.828575 сек
Связь с ML: Операции с тензорами в нейронных сетях сводятся к умножению матриц (BLAS Level 3). Пример: свёртка в CNN → преобразование в dgemm.
Реальный кейс: PyTorch использует BLAS через MKL или OpenBLAS для ускорения обучения моделей.
Интересно:
NVIDIA cuBLAS переносит BLAS на GPU, увеличивая скорость в 10–100 раз.
OpenBLAS адаптируется под ARM (например, Apple M1/M2).

Что такое LAPACK?

LAPACK (Linear Algebra Package) — библиотека для сложных задач линейной алгебры:

- Решение систем линейных уравнений (Ax = b).
- Разложение матриц (LU, QR, SVD).
- Вычисление собственных значений и векторов.

Пример: Решение Ax = b через LU-разложение:

Реализация:

Референсная версия: http://www.netlib.org/lapack/

Оптимизированная: MKL, OpenBLAS (с LAPACK внутри).

Интересное про LAPACK — связь с BLAS

LAPACK делегирует базовые операции (умножение матриц, сложение) BLAS.

Эффективность LAPACK зависит от скорости BLAS (Level 3).

Пример оптимизации: LU-разложение в dgesv использует dgemm (BLAS) для матричных операций:

A = P * L * U, где P — матрица перестановок

OpenBLAS/MKL ускоряют dgemm, ускоряя весь LAPACK.

Интересный факт: LAPACK изначально писался на Fortran 77 (1992 год), но до сих пор актуален благодаря BLAS и новым реализациям (LAPACKE для C).

Установка LAPACK

OpenBLAS включает LAPACK, что упрощает установку.

Установка на Ubuntu:

```
sudo apt update
sudo apt install liblapacke-dev
```

- • Компиляция из исходников:

1. Скачать с GitHub:

```
git clone https://github.com/OpenMathLib/OpenBLAS.git
```

2. Сборка:
3. cd OpenBLAS
4. make

```
make USE_OPENMP=1 LAPACK=1
```

5. Библиотека обычно устанавливается в /usr/local/lib.

- Проверка: После установки доступны файлы libopenblas.a и libopenblas.so.
- Связывание с C:

```
gcc -o program program.c -I/usr/local/include -L/usr/local/lib -lopenblas
```

Пример кода с LAPACK

Задача: Решить систему Ax = b с помощью dgesv (LU-разложение).

```c
#include <stdio.h>
#include <lapacke.h>
int main() {
    lapack_int n = 3, nrhs = 1, lda = 3, ldb = 3, info;
    lapack_int ipiv[3];                          // Для перестановок
    double A[9] = {4, 1, 1, 2, 5, 2, 1, 1, 3}; // Матрица 3x3
    double b[3] = {1, 2, 3};                     // Вектор b
    // Решаем систему
    info = LAPACKE_dgesv(LAPACK_ROW_MAJOR, n, nrhs, A, lda, ipiv, b, ldb);
    if (info == 0)
        printf("Решение: x = [%f, %f, %f]\n", b[0], b[1], b[2]);
    else
        printf("Ошибка: %d\n", info);
    return 0;
}
```

```
$ gcc main.c -o main -llapacke -lopenblas
$ ./main
Собственные значения: [2.000000, 3.267949, 6.732051]
```

Пример 2 кода с LAPACK

Задача: Найти собственные значения матрицы с помощью dsyev (симметричная матрица).

```c
#include <stdio.h>
#include <lapacke.h>
int main() {
    lapack_int n = 3, lda = 3, info;
    double A[9] = {5, 2, 1, 2, 4, 0, 1, 0, 3}; // Симметричная 3x3
    double w[3];                                // Собственные значения
    info = LAPACKE_dsyev(LAPACK_ROW_MAJOR, 'N', 'U', n, A, lda, w);
    if (info == 0)
        printf("Собственные значения: [%f, %f, %f]\n", w[0], w[1], w[2]);
    else
        printf("Ошибка: %d\n", info);
    return 0;
}
```

```
$ gcc main.c -o main -llapacke -lopenblas
$ ./main
Решение: x = [0.222222, 2.000000, 3.000000]
```

Что такое Intel MKL?

MKL (Math Kernel Library) — библиотека от Intel для высокопроизводительных вычислений:

- Включает BLAS и LAPACK с оптимизацией под процессоры Intel.
- Дополнительно: FFT (быстрое преобразование Фурье), sparse-операции, генерация случайных чисел.

Особенности:

- Поддержка многопоточности (OpenMP) и векторных инструкций (AVX, AVX-512).
- Интеграция с C, C++, Fortran, Python.

Интересное про MKL

Преимущества:

- Оптимизация под Intel CPU (AVX-512 даёт до 2x ускорения).
- Поддержка GPU (oneAPI) для будущих расширений.
- Интеграция с инструментами Intel (VTune, Advisor).

Интересный факт:

MKL автоматически выбирает лучший алгоритм в зависимости от размера матрицы и процессора.

Сравнение производительности:

OpenBLAS vs MKL на Intel Core i7: MKL до 20% быстрее для больших матриц.

Установка Intel MKL

Установка MKL на Ubuntu

Установка через apt:

Добавьте GPG-ключ:

```
wget -O- https://apt.repos.intel.com/intel-gpg-keys/GPG-PUB-KEY-INTEL-SW-PRODUCTS.PUB | gpg --dearmor | sudo tee /usr/share/keyrings/oneapi-archive-keyring.gpg > /dev/null
```

Добавьте репозиторий:

```
echo "deb [signed-by=/usr/share/keyrings/oneapi-archive-keyring.gpg] https://apt.repos.intel.com/oneapi all main" | sudo tee /etc/apt/sources.list.d/oneAPI.list
```

Обновите и установите:

```
sudo apt update
sudo apt install intel-oneapi-mkl
sudo apt install intel-oneapi-mkl-devel
```

Настройте окружение:

```
source /opt/intel/oneapi/setvars.sh
```

Проверка: Библиотеки в /opt/intel/oneapi/mkl/latest/lib.

Связывание с C:

```
gcc -o program program.c -I/opt/intel/oneapi/mkl/latest/include -L/opt/intel/oneapi/mkl/latest/lib/intel64 -lmkl_intel_lp64 -lmkl_core -lmkl_sequential -lpthread -lm
```

Пример кода с MKL (BLAS)

Задача: Умножить матрицы A и B с помощью cblas_dgemm.

```c
#include <stdio.h>
#include <mkl.h>
int main()
{
    double A[] = {1, 2, 3, 4, 5, 6};     // 2x3
    double B[] = {7, 8, 9, 10, 11, 12}; // 3x2
    double C[4] = {0};                   // 2x2
    double alpha = 1.0, beta = 0.0;
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                2, 2, 3, alpha, A, 3, B, 2, beta, C, 2);
    printf("C = [%f, %f; %f, %f]\n", C[0], C[1], C[2], C[3]);
    return 0;
}
```

```
$ gcc main.c -o main -I/opt/intel/oneapi/mkl/latest/include -L/opt/intel/oneapi/mkl/latest/lib/intel64 -lmkl_intel_lp64 -lmkl_core -lmkl_sequential -lpthread -lm
$ ./main
C = [58.000000, 64.000000; 139.000000, 154.000000]
```

Пример кода с MKL (LAPACK)

Задача: Решить систему Ax = b с помощью dgesv.

```c
#include <stdio.h>
#include <mkl.h>
int main() {
    MKL_INT n = 3, nrhs = 1, lda = 3, ldb = 3, info;
    MKL_INT ipiv[3];
    double A[9] = {4, 1, 1, 2, 5, 2, 1, 1, 3};
    double b[3] = {1, 2, 3};
    info = LAPACKE_dgesv(LAPACK_ROW_MAJOR, n, nrhs, A, lda, ipiv, b, ldb);
    if (info == 0)
        printf("Решение: x = [%f, %f, %f]\n", b[0], b[1], b[2]);
    else
        printf("Ошибка: %d\n", info);
    return 0;
}
```

```
$ gcc main.c -o main -I/opt/intel/oneapi/mkl/latest/include -L/opt/intel/oneapi/mkl/latest/lib/intel64 -lmkl_intel_lp64 -lmkl_core -lmkl_sequential -lpthread -lm
$ ./main
Решение: x = [0.222222, 2.000000, 3.000000]
```

План презентации

Объектные файлы	BLAS, LAPACK, MKL	История и архитектура Unix
35 минут	20 минут	35 минут

Введение

Unix — первая операционная система, реализованная на достаточно высокоуровневом языке программирования, C, который, в свою очередь, был разработан специально для этой цели и получил известность и влияние благодаря Unix. Хотя, конечно, стоит отметить, что C больше не считается языком программирования высокого уровня.

Если бы в 1970-е и 1980-е годы инженеры в Bell Labs решили перейти с C на другой язык для разработки новой версии Unix, то мы бы сегодня говорили именно об этом.

Высказывание Денниса М. Ритчи, одного из пионеров C, о роли Unix в успехе данного языка:

«Успех самой системы Unix, несомненно, сыграл самую важную роль; благодаря ему этот язык стал доступен сотням тысяч людей. С другой стороны, конечно, использование C в Unix обеспечило переносимость этой системы на широкий спектр компьютеров, что было важно для ее успеха.»

История Unix

Multics OS и Unix

Еще до Unix существовала система ***Multics OS*** — совместный проект MIT, General Electric и Bell Labs, запущенный в 1964 году. Это был первый в мире пример рабочей и безопасной операционной системы, что обеспечило ее успех. Multics устанавливали везде, от университетов до правительственных учреждений. Если перенестись обратно в наше время, то все современные ОС заимствуют те или иные идеи из Multics.

В 1969 году по разным причинам, часть сотрудников Bell Labs, особенно такие пионеры, как Кен Томпсон и Деннис Ритчи, разочаровались в Multics, в результате чего проект был заброшен. Компания Bell Labs разработала собственную, более простую и эффективную операционную систему - Unix.

Вдобавок стоит отметить, что компания Bell Labs работала над новой распределенной операционной системой под названием Plan 9, которая была основана на проекте Unix.

Полагаю, достаточно сказать, что система Unix являлась упрощенным выражением идей и инноваций, представленных в Multics; она не была чем-то новым.

Сходства и различия Unix и Multics.

- Внутренняя структура обеих ОС имеет многослойную архитектуру. Это значит, их архитектура состоит примерно из одних и тех же колец. Особенно это касается ядра и командной оболочки. Кроме того, Unix и Multics предоставляют ряд собственных утилит, таких как ls и pwd.
- Системе Multics для работы требовались дорогие ресурсы и оборудование. Это был один из основных недостатков, который открыл дорогу.
- Архитектура Multics была сложной. В этом состояла основная причина разочарования работников Bell Labs. Систему Unix изначально пытались сделать простой. В первой версии не было даже многозадачности и многопользовательского режима!

BCPL и B

Язык программирования BCPL был создан Мартином Ричардсом для написания компиляторов. Сотрудники Bell Labs познакомились с этим языком во время работы над Multics. После закрытия проекта компания Bell Labs начала писать Unix на ассемблере. В те времена это был единственный приемлемый вариант!

Факт, что участники проекта Multics использовали в разработке PL/1, считался чем-то необычным, но являлся доказательством того, что операционную систему можно успешно написать на языке программирования высокого уровня. И это стало основной причиной, почему разработка Unix была переведена на другой язык.

Кен Томпсон и Деннис Ритчи продолжили попытки написания модулей операционной системы на языке, отличном от ассемблера. Они попробовали использовать BCPL, но оказалось, что для применения этого языка на компактных компьютерах наподобие DEC PDP-7 его необходимо модифицировать. Эти модификации привели к появлению языка программирования B.

Язык B, будучи системным языком программирования, имел недостатки. В нем не было системы типов, из-за чего в каждой операции он позволял работать только с машинными словами (а не с байтами). Это затрудняло его применение на компьютерах с разной длиной машинных слов.

Язык B со временем претерпел ряд изменений, которые привели к появлению языка NB (new B — «новый B»), унаследовавшего структуры из B. В языке B эти структуры не имели типа, но в C стали типизированными. Наконец, в 1973 году вышла четвертая версия Unix, написанная с использованием C + assembler.

Путь к C

Недостатки B, которые привели к появлению C.

- Язык B позволял работать только с машинными словами в памяти.
- Язык B не поддерживал типы. Если более точно, то B был однотипным языком.
- Отсутствие типов привело к тому, что многие операции, ориентированные на байты (такие как алгоритмы для работы со строками), были неэффективно написаны на B. Причина — язык B использовал для работы с памятью слова, а не байты, что не позволяло эффективно управлять многобайтными типами данных, такими как целые числа и строки.
- Язык B не поддерживал операции с плавающей запятой. В то время подобные операции становились все более доступными в новом оборудовании, но их поддержка в языке B отсутствовала.
- Несмотря на существование таких компьютеров, как PDP-1, которые могли работать с памятью побайтно, язык B демонстрировал низкую эффективность при адресации байтов памяти.

Недостатки языка B вынудили Денниса Ритчи создать новый язык, NB (new B), который в итоге превратился в C.

Этот новый язык пытался исправить изъяны и трудности, присущие B, и стал фактически стандартом в мире системного программирования, заменив ассемблер. Менее чем через десять лет новые версии Unix уже были полностью написаны на C, и с тех пор все ОС, основанные на Unix, полагаются на данный язык и его важнейшую роль в системе.

На сегодня у него нет конкурентов!

В отличие от многих других языков программирования C является стандартом ISO.

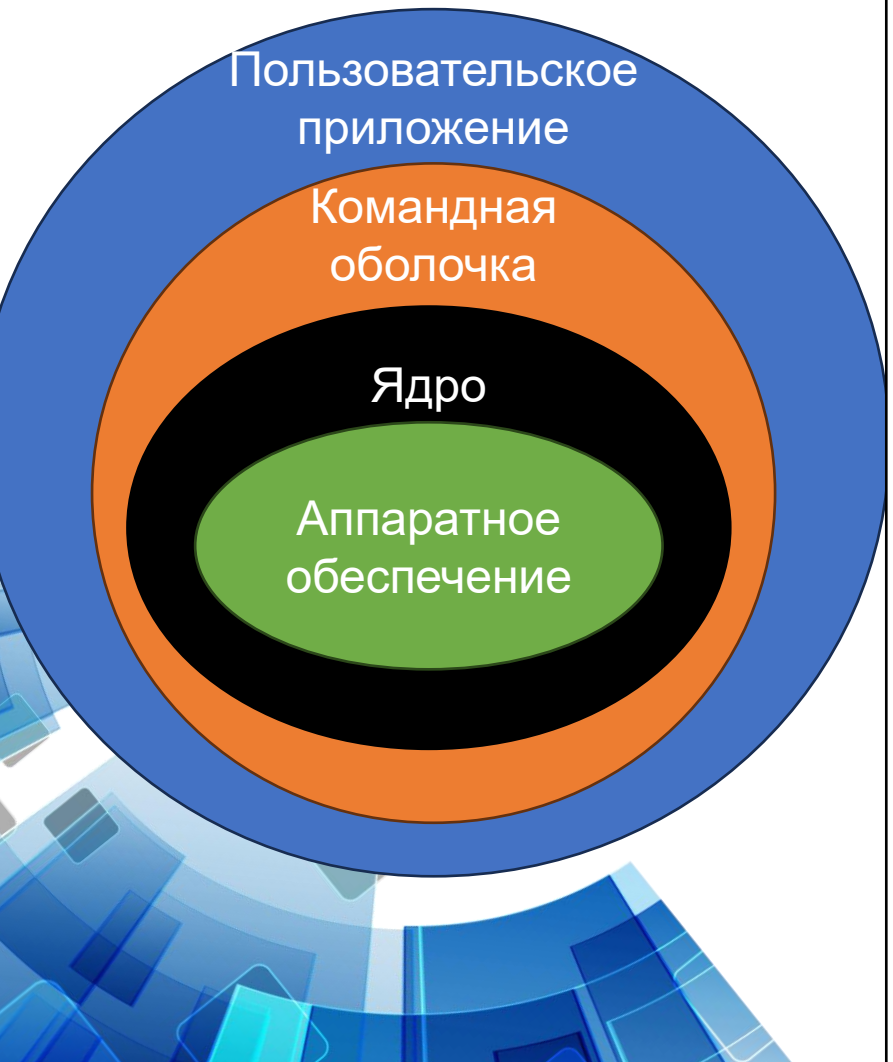
Архитектура Unix

Философия

- Система Unix проектировалась и разрабатывалась в основном для программистов, а не для обычных пользователей. В связи с этим многие требования, относящиеся к пользовательскому интерфейсу и взаимодействию с пользователями, не являются частью архитектуры Unix.
- Unix состоит из множества мелких и простых программ. Каждая из них предназначена для выполнения небольшой и простой задачи. Существует множество примеров таких программ, включая ls, mkdir, ifconfig, grep и sed.
- Unix поощряет программистов разбивать свои большие и сложные проекты на мелкие и простые программы.
- Каждая мелкая и простая программа должна уметь подавать свой вывод на вход другой программе, продолжая цепочку выполнения. Таким образом, мелкие программы можно объединять в цепочку для выполнения сложных задач. Например, ls -l | grep a.out.
- Система Unix строго ориентирована на работу с текстом. Вся конфигурация хранится в текстовых файлах, и командная строка тоже текстовая. Скрипты командной оболочки тоже представляют собой текстовые файлы с простым синтаксисом для написания алгоритмов.
- Unix поощряет выбор простоты перед совершенством.
- Программы, написанные для определенной операционной системы, совместимой с Unix, должны легко переноситься на другие системы данного семейства. Это в основном достигается благодаря наличию единой кодовой базы.

Какую роль в этом играет язык C? Дело в том, что почти все ключевые элементы (то есть мелкие и простые программы, лежащие в основе Unix), о которых шла речь выше, написаны на данном языке. Пример: исходный код программы ls из NetBSD.

Многослойная структура Unix



Основная задача операционной системы — дать пользователю возможность взаимодействовать с оборудованием.

Одна из важнейших задач, стоящих перед Unix: сделать оборудование доступным для программ, которые хотят к нему обращаться.

Ядро - это самая важная часть операционной системы. Оно ближе всего находится к оборудованию и служит оберткой для предоставления возможностей, которыми обладает данное оборудование.

Командная оболочка – это обертка, которая позволяет пользовательским приложениям взаимодействовать с ядром и применять его многочисленные функции. Оно также содержит набор библиотек, полностью написанных на C, с помощью которых программист может разрабатывать новые приложения для Unix.

Используя библиотеки, описанные в спецификации SUS (Simple Unix Specification — простая спецификация Unix), кольцо командной оболочки должно предоставлять программистам стандартный и строго определенный интерфейс.

Пользовательские приложения – кольцо, состоящее из прикладных программ, предназначенных для выполнения в системах Unix. Речь идет о базах данных, веб-сервисах, почтовых серверах, браузерах, электронных таблицах и текстовых процессорах. Эти приложения должны использовать API и инструменты, предоставляемые командной оболочкой, не обращаясь к ядру напрямую (через системные вызовы, которые мы вскоре обсудим).

Характерная черта архитектуры Unix — тот факт, что внешние кольца должны взаимодействовать с внутренними через определенный интерфейс.

Интерфейс командной оболочки для ПП

Чтобы применить возможности, доступные в системе Unix, живой пользователь работает либо с терминалом, либо с отдельной графической программой, такой как браузер. И то и другое принадлежит к категории пользовательских приложений. Оперативная память, центральный процессор, сетевой адаптер и жесткие диски — все это типичные примеры аппаратного обеспечения, которое большинство Unix-программ задействуют через API кольца командной оболочки.

С точки зрения разработчика, приложение мало чем отличается от программы. Но в понимании пользователя приложение — программа, с которой можно взаимодействовать через графический или консольный интерфейс (GUI и CLI соответственно), а программа — просто программный компонент, который выполняется на компьютере без какого-либо пользовательского интерфейса (как в случае с сервисом).

Как написать такую программу на C, которую можно собрать в разных версиях Unix и на разных аппаратных платформах?

Ответ прост: все системы Unix предоставляют для своей командной оболочки один и тот же интерфейс прикладного программирования (Application Programming Interface, API). Фрагмент кода на языке C, который использует только этот стандартный интерфейс, можно собрать и запустить на любой Unix-системе.

API — куча заголовочных файлов с объявлениями.

На самом деле заголовок stdio.h не является частью C, несмотря на то что он и объявленные в нем функции фигурируют во всех книгах о данном языке. Это часть стандартной библиотеки C, описанной в стандарте SUS. Программе, написанной на C для Unix, ничего не известно о реализации тех или иных функций, таких как printf или fopen. Иными словами, программы во внешнем кольце воспринимают командную оболочку как некий черный ящик.

Интерфейс командной оболочки для ПП

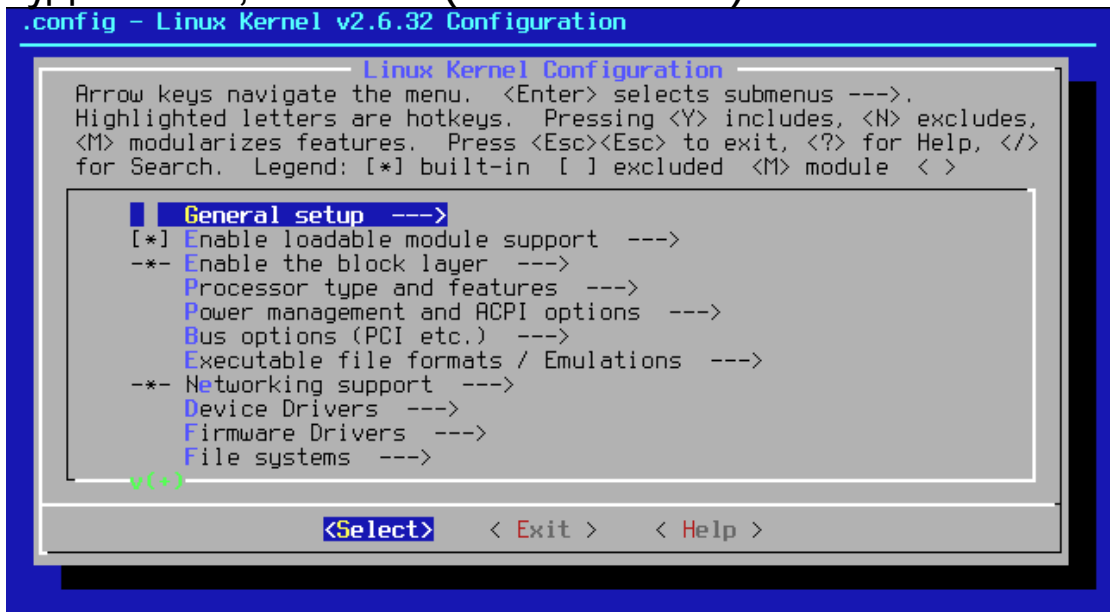
В стандарте SUS собраны различные API, предоставляемые кольцом командной оболочки. Этот стандарт развивается консорциумом The Open Group и имеет несколько версий, вышедших с момента создания Unix.

- ***Системные интерфейсы*** — сюда входят все функции, которыми могут пользоваться любые программы на языке C. В стандарте SUS v4 предусмотрена 1191 функция, которая должна быть реализована в Unix-системе.
- **Заголовочные интерфейсы** — список заголовочных файлов, которые могут быть доступны в Unix-системе, совместимой с SUS v4. В этой версии SUS перечислены 82 заголовочных файла, к которым может обращаться любая программа на языке C. Если пройтись по этому списку, то можно найти много известных заголовков, таких как stdio.h, stdlib.h, math.h и string.h.
- ***Вспомогательные интерфейсы*** — список консольных утилит или программ командной строки, которые должны быть доступны в Unix-системе, совместимой с SUS v4. Если пройтись по таблицам, то можно встретить много знакомых нам команд, таких как mkdir, ls, cp, df, bc; всего их насчитывается 160.

- ***Сценарный интерфейс*** — язык, который используется для написания скриптов командной оболочки. С его помощью в основном автоматизируют задачи, в которых применяются консольные утилиты. Этот интерфейс обычно называют языком командной оболочки (или командной строки).
- ***Интерфейсы XCURSES*** — набор интерфейсов, которые позволяют программе, написанной на C, взаимодействовать с пользователем с помощью минималистичного текстового GUI.

Интерфейс командной оболочки для ПП

В SUS v4 интерфейсу XCURSES отводится 379 функций, размещенных в трех заголовках, а также четыре утилиты.
Многие программы до сих пор применяют XCURSES для более удобного взаимодействия с пользователем. Следует отметить, что интерфейсы на основе XCURSES не нуждаются в графической подсистеме. Благодаря этому с ними можно работать удаленно, по SSH (Secure Shell).



Конфигурационное меню на основе ncurses

Интерфейс командной оболочки для ПП

SUS не описывается иерархия файловой системы и то, где можно найти те или иные заголовочные файлы. Этот стандарт лишь указывает на то, какие заголовки должны присутствовать и быть доступны в системе. Согласно широко распространенному соглашению о заголовочных файлах они должны находиться либо в /usr/include, либо /usr/local/include, но окончательное решение по-прежнему остается за операционной системой и пользователем.

libc — это набор функций, размещенных в определенных заголовочных файлах в соответствии с SUS, плюс статические и динамические библиотеки с реализациями доступных функций.

Определение libc тесно связано с процессом стандартизации систем Unix. Любая программа на языке C, разработанная в системе Unix, использует libc для взаимодействия с более низкими уровнями ядра и аппаратного обеспечения.

Не все операционные системы полностью совместимы с Unix. Например, Microsoft Windows и ОС с ядром Linux (Android).

Unix-совместимая система полностью соответствует стандартам SUS, чего нельзя сказать о Unix-подобной системе, которая имеет лишь частичную совместимость.

Появление множества Unix-подобных операционных систем, особенно после рождения Linux, создало необходимость в классификации этого конкретного подмножества SUS. Данный стандарт был назван POSIX (Portable Operating System Interface — переносимый интерфейс операционных систем). Можно сказать, что POSIX — подмножество SUS, с которым совместимы Unix-подобные системы.

Unix-подобные ОС, включая большинство дистрибутивов Linux, изначально POSIX-совместимы. Вот почему с Ubuntu можно работать так же, как с FreeBSD.

Интерфейс ядра для кольца КО

Нам нужен был процесс компиляции и механизм компоновки, чтобы спроектировать операционную систему, которая предоставляет интерфейс и реализует набор библиотечных файлов. Вы уже должны заметить, что каждая возможность языка С играет на руку Unix. Чем лучше вы будете понимать отношения между С и Unix, тем более очевидной будет для вас их тесная связь.

Для работы с ядром libc (или функции в кольце командной оболочки) в основном используют системные вызовы. Чтобы объяснить этот механизм и показать, в каком месте многослойной модели применяются системные вызовы, нам нужен реальный пример.

Нам также следует выбрать конкретную реализацию libc, чтобы иметь возможность проанализировать исходники и найти соответствующие системные вызовы. Например, FreeBSD. Это Unix-подобная операционная система, которая является ответвлением от BSD Unix. Исходники libc этой системы можно найти в директории lib/libc.

подключается функция sleep из кольца ядра

```c
#include <unistd.h>
int main(int argc, char **argv)
{
    sleep(1);
    return 0;
}
```

Код подключает заголовочный файл unistd.h и вызывает функцию sleep; и то и другое — часть интерфейсов, доступных в SUS. Но что происходит дальше, особенно в функции sleep? Системные вызовы инициируются кодом, написанным в реализации libc. На самом деле именно так вызываются процедуры ядра. В SUS и впоследствии в POSIX-совместимых системах была предусмотрена программа, которая позволяла отслеживать системные вызовы во время выполнения кода.

Интерфейс ядра для кольца КО

```
$ gcc main.c -o a.out
$ truss ./a.out //для FreeBSD
$ strace ./a.out //аналог, если нет в вашей ОС truss
mmap(0x0,32768,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANON,-1,0x0) = 34366160896
(0x800620000)
issetugid() = 0 (0x0)
...
openat(AT_FDCWD,"/lib/libc.so.7",O_RDONLY|O_CLOEXEC|O_VERIFY,00) = 3 (0x3)
...
nanosleep({ 1.000000000 }) = 0 (0x0)
sigprocmask(SIG_BLOCK,{ SIGHUP|SIGINT|SIGQUIT|SIGKILL|SIGPIPE|SIGALRM|
SIGTERM|SIGURG|SIGSTOP|SIGTSTP|SIGCONT|SIGCHLD|SIGTTIN|SIGTTOU|SIGIO|
SIGXCPU|SIGXFSZ|SIGVTALRM|SIGPROF|SIGWINCH|SIGINFO|SIGUSR1|
SIGUSR2 },{ }) = 0 (0x0)
...
sigprocmask(SIG_SETMASK,{ },0x0) = 0 (0x0)
exit(0x0)
process exit, rval = 0
```

В этом простом примере сделано много системных вызовов. Некоторые из них относятся к загрузке разделяемых объектных библиотек, особенно на этапе инициализации процесса. Полужирным шрифтом, открывает разделяемый объектный файл libc.so.7, содержащий непосредственную реализацию libc для FreeBSD; программа делает системный вызов nanosleep.

Интерфейс ядра для кольца КО

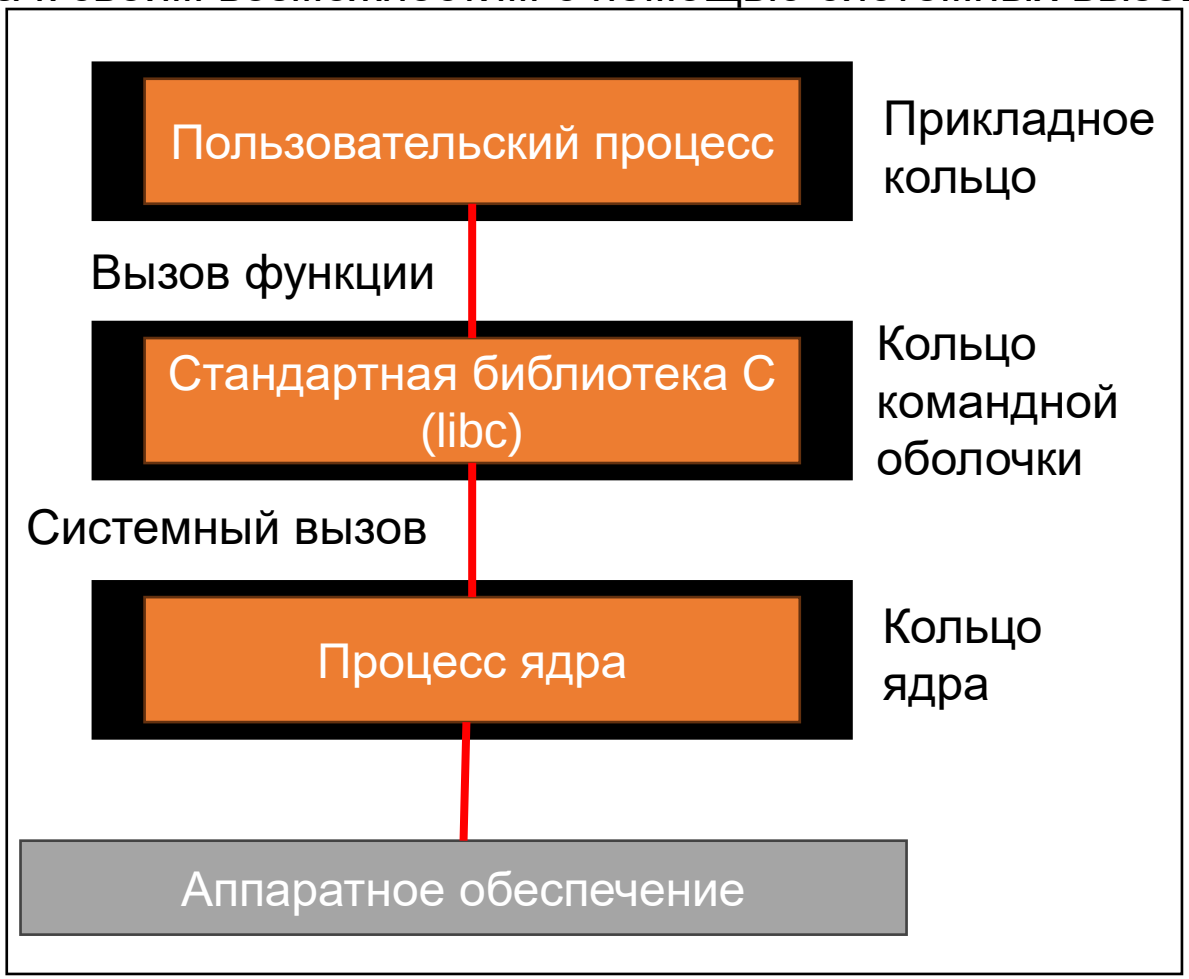
Теперь найдем то место в libc, где инициируется системный вызов.
Если зайти в каталог lib/libc и выполнить grep для sys_nanosleep, увидим следующие файлы:

```
$ git clone https://github.com/freebsd/freebsd-src.git
...
$ cd freebsd-src
$ git reset --hard bf78455d496
...
$ cd lib/libc
$ grep sys_nanosleep . -R
./include/libc_private.h:int            __sys_nanosleep(const struct timespec *,
struct timespec *);
./sys/interposing_table.c:      SLOT(nanosleep, __sys_nanosleep),
./sys/nanosleep.c:__weak_reference(__sys_nanosleep, __nanosleep);
./sys/Symbol.map:       __sys_nanosleep;
```

Как можно видеть в файле lib/libc/sys/interposing_table.c, функция nanosleep привязана к вызову __sys_nanosleep. Следовательно, при выполнении nanosleep всегда вызывается __sys_nanosleep.
В FreeBSD названия всех функций, которые являются системными вызовами, принято начинать с __sys.
В приведенном выше терминале есть еще один интересный участок. Файл lib/ libc/include/libc_private.h содержит объявления приватных и внутренних функций-оберток вокруг системных вызовов.

Ядро

Основное назначение кольца ядра состоит в управлении оборудованием, подключенным к системе, и предоставлении доступа к своим возможностям с помощью системных вызовов.



Ядро

- Процесс ядра — первое, что загружается и выполняется в системе. Только вслед за ним можно запускать пользовательские процессы.
- Процесс ядра существует в единственном экземпляре, при этом мы можем запустить сразу несколько пользовательских процессов.
- Процесс ядра создается в результате того, что загрузчик копирует его образ в основную память, а для создания пользовательского процесса применяются системные вызовы exec оли fork, присутствующие в большинстве Unix-систем.
- Процесс ядра обрабатывает и выполняет системные вызовы, в то время как пользовательский их инициирует и ждет их выполнения.
- Процесс ядра имеет привилегированный доступ к физической памяти и всему подключенному оборудованию, в то время как пользовательский работает с виртуальной памятью, которая является отражением части физической памяти, и ничего не знает о ее физической структуре. Пользовательский процесс имеет управляемый и наблюдаемый доступ к ресурсам и оборудованию.

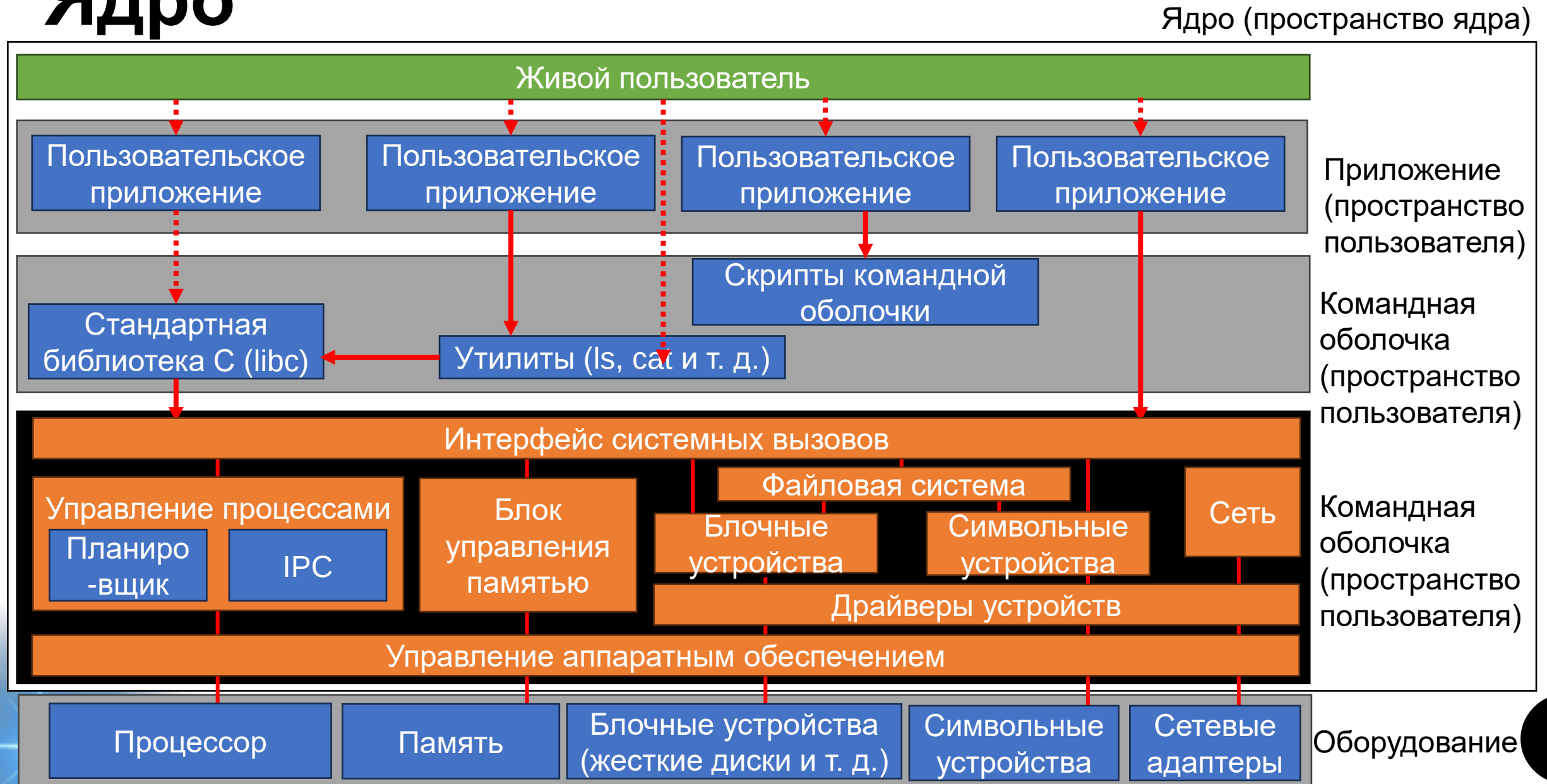
Среда выполнения операционной системы имеет два разных режима. Один предназначен для процесса ядра, а другой — для пользовательских процессов. Первый режим выполнения называется пространством ядра, а второй — пользовательским пространством..

Ядро

Внутреннюю структуру типичного Unix-ядра можно разделить по выполняемым задачам:

- ***Управление процессами.*** Ядро создает пользовательские процессы с помощью системного вызова. Среди прочих операций перед запуском нового процесса необходимо сначала выделить память и загрузить его инструкции.
- ***Межпроцессное взаимодействие*** (Inter-Process Communication, IPC). Пользовательские процессы, запущенные на одном компьютере, могут задействовать разные методы обмена данными, включая разделяемую память, каналы и доменные сокеты Unix.
- ***Планирование.*** Ядро управляет доступом к центральному процессору, пытаясь его балансировать. **Планирование** — процедура разделения процессорного времени между разными процессами в зависимости от их приоритета и значимости.
- ***Управление памятью.*** Ядро — единственный процесс, который видит всю физическую память и имеет к ней неограниченный доступ. Поэтому на него ложится разбиение ее на выделяемые страницы, назначение новых страниц процессам (при выделении кучи), освобождение памяти и многие другие сопутствующие процедуры.
- ***Начальная загрузка системы.*** После загрузки образа ядра в основную память его процесс должен запуститься и инициализировать пользовательское пространство. Обычно для этого создается первый пользовательский процесс с PID.
- ***Управление устройствами.*** Ядро должно иметь возможность управлять не только процессором и памятью, но и аппаратным обеспечением. Для этого предусмотрен слой абстракции. В Unix устройства привязываются к файлам, которые обычно хранятся в каталоге /dev.

Ядро


Ядро (пространство ядра)
Живой пользователь
Пользовательское приложение
Пользовательское приложение
Пользовательское приложение
Пользовательское приложение
Приложение (пространство пользователя)
Скрипты командной оболочки
Стандартная библиотека C (libc)
Утилиты (ls, cat и т. д.)
Командная оболочка (пространство пользователя)
Интерфейс системных вызовов
Управление процессами
Планиро -вщик
IPC
Блок управления памятью
Файловая система
Блочные устройства
Символьные устройства
Сеть
Драйверы устройств
Управление аппаратным обеспечением
Командная оболочка (пространство пользователя)
Процессор
Память
Блочные устройства (жесткие диски и т. д.)
Символьные устройства
Сетевые адаптеры
Оборудование

Аппаратное обеспечение

```
$ ls -l /dev
total 0
crw-r--r-- 1 root root     10, 235 Mar  2 15:24 autofs
drwxr-xr-x 2 root root         580 Mar  2 15:24 block
drwxr-xr-x 2 root root         100 Mar  2 15:24 bsg
crw-rw---- 1 root disk     10, 234 Mar  2 15:24 btrfs-control
drwxr-xr-x 3 root root          60 Mar  2 15:24 bus
drwxr-xr-x 2 root root        2740 Mar  2 15:24 char
crw--w---- 1 root tty        5,   1 Mar  2 15:24 console
...
lrwxrwxrwx 1 root root          28 Mar  2 15:24 log -> /run/systemd/journal/dev-log
crw-rw---- 1 root disk     10, 237 Mar  2 15:24 loop-control
...
lrwxrwxrwx 1 root root          15 Mar  2 15:24 stderr -> /proc/self/fd/2
lrwxrwxrwx 1 root root          15 Mar  2 15:24 stdin -> /proc/self/fd/0
lrwxrwxrwx 1 root root          15 Mar  2 15:24 stdout -> /proc/self/fd/1
crw-rw-rw- 1 root tty        5,   0 Mar  2 15:27 tty
...
drwxr-xr-x 2 root root          60 Mar  2 15:24 vfio
crw-rw---- 1 root kvm      10, 238 Mar  2 15:24 vhost-net
drwxr-xr-x 2 root root          80 Mar  2 15:24 virtio-ports
```

Благодаря слою абстракции поверх аппаратного обеспечения система Unix поддерживает виртуальные устройства.