Last updated: 2020-09-01
vadimov@i.ua

Занятие #2.

Краткая теоретическая справка

Сегменты

Объектные файлы и исполняемые файлы имеют один из нескольких различных форматов. В большинстве реализаций Linux этот формат называется ELF (первоначально «Extensible Linker Format», теперь «Executable and Linking Format»). В других системах исполняемый формат - COFF (Common Object-File Format). Вы можете узнать больше о формате, используемом в системе UNIX, набрав man a.out и прочитав справочную страницу.

Все эти разные форматы имеют общую концепцию сегментов. Что касается объектных файлов, то это просто области в двоичном файле, где хранится вся информация определенного типа (например, записи таблицы символов). Также широко используется термин «раздел»; разделы - это наименьшая единица организации в файле ELF. Сегмент обычно состоит из нескольких разделов. Не путайте понятие сегмента в UNIX с концепцией сегмента в архитектуре Intel x86.

- Сегмент в UNIX - это раздел связанного материала в двоичном файле.
- Сегмент в модели памяти Intel x86 является результатом конструкции, в которой (по соображениям совместимости) адресное пространство не является однородным, а разделено на 64-килобайтные диапазоны, известные как сегменты.

Тема сегментов архитектуры Intel x86 заслуживает отдельного обсуждения. Когда вы запускаете size для исполняемого файла, он сообщает вам размер трех сегментов, известных как текст, данные и bss в исполняемом файле. Например:

```
Vadim@Stoic ~/hspl/pract2
$ gcc -Wall hello.c

Vadim@Stoic ~/hspl/pract2
$ ls -l
total 169
-rwxr-xr-x 1 Vadim None 167993 Aug 25 20:46 a.exe
-rw-r--r-- 1 Vadim None     85 Aug 25 20:46 hello.c
```

`size` не печатает заголовки, поэтому используйте эхо для их создания:

```
Vadim@Stoic ~/hspl/pract2
$ echo; echo "text data bss total" ; size a.exe

text data bss total
   text    data     bss     dec     hex filename
   2989    1956     416    5361    14f1 a.exe
```

Еще один способ проверить содержимое исполняемого файла - это использовать утилиты `nm` или `dump`. Скомпилируйте исходный код ниже и запустите nm на полученном `a.out`.

```
Vadim@Stoic ~/hspl/pract2
```

```
$ cat fruts.c
#include <stdio.h>
#include <stdlib.h>

char pear[40];
static double peach;
int mango = 13;
static long melon = 2001;

int main() {
    int i=3, j, *ip;
    ip = malloc(sizeof(i));
    pear[5] = i;
    peach = 2.0*mango;
    return 0;
}
Vadim@Stoic ~/hspl/pract2
$ gcc fruts.c

Vadim@Stoic ~/hspl/pract2
Vadim@Stoic ~/hspl/pract2
$ nm -... a.exe
. . .
```

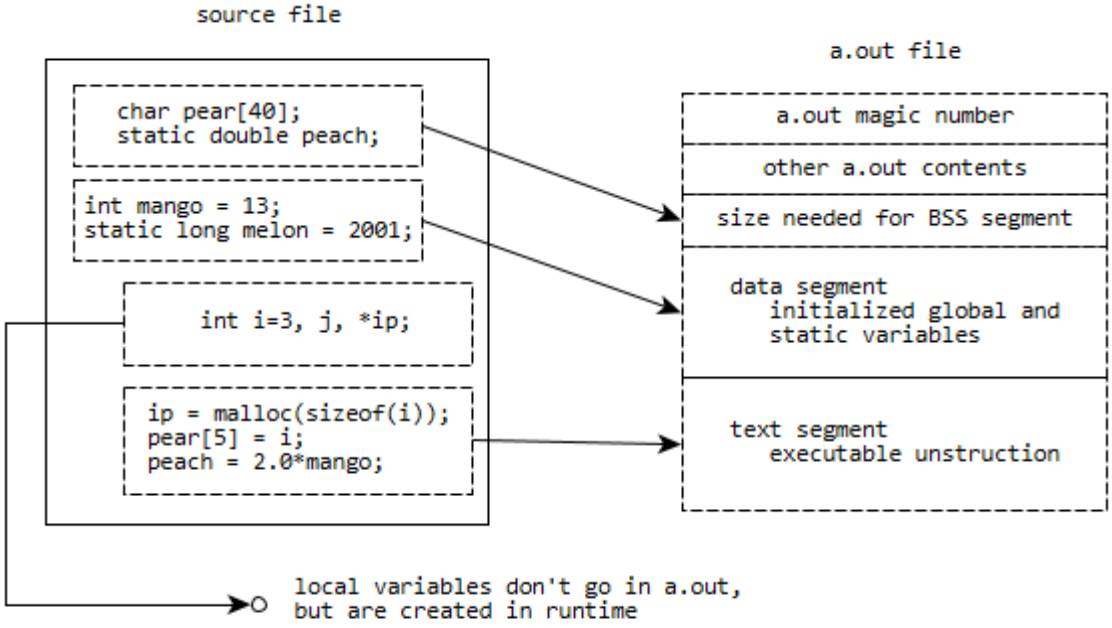


Fig shows what the compiler/linker puts in each segment

Известный факт: в программе на C *неинициализированные* глобальные и статические переменные автоматически инициализируются нулем. А как *инициализированные* глобальные объекты? Инициализированный сегмент данных - это область адресного пространства, где хранятся явно инициализированные глобальные и статические переменные.

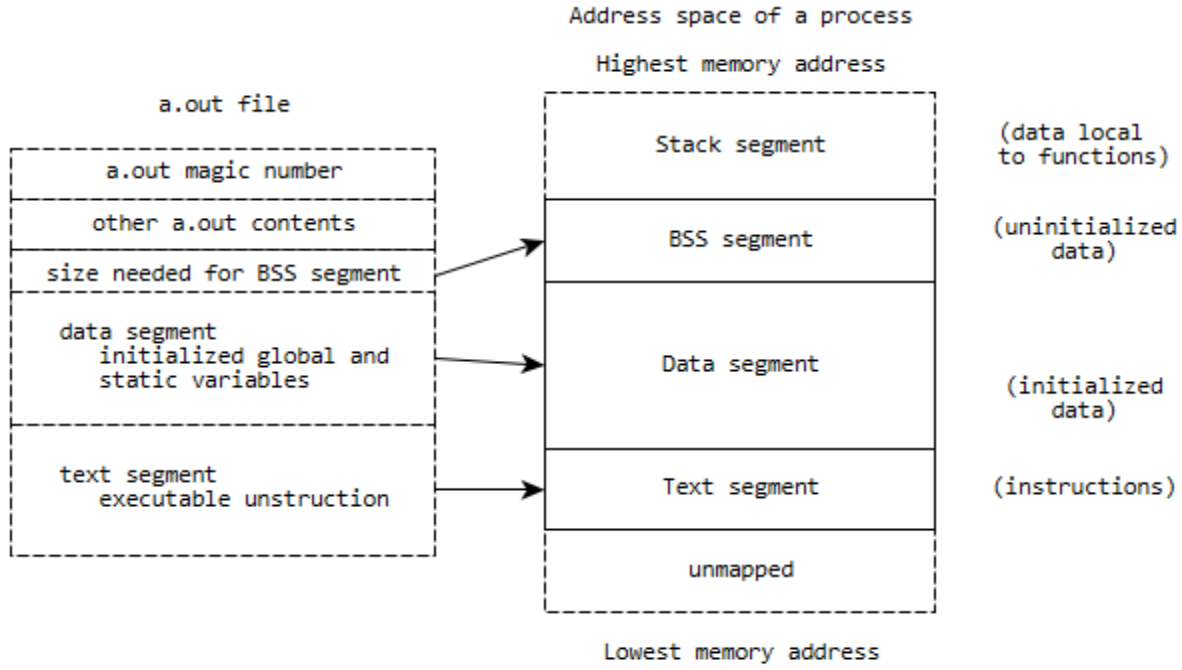
Сегмент неинициализированных данных - это область адресного пространства, в которой находятся неинициализированные глобальные переменные и статические переменные. На самом деле они неявно инициализируются нулем (memset нуль). Запомните, что в более старой литературе этот сегмент часто упоминается как BSS !

Сегмент BSS получил свое название от аббревиатуры «Block Started by Symbol» - псевдооперации из старого ассемблера IBM 704, перенесенного в UNIX еще и с тех пор. Некоторым нравится запоминать его как «Better Save Space». Поскольку сегмент BSS содержит

только переменные, которые еще не имеют никакого значения. Размер, который потребуется BSS во время выполнения, записывается в объектном файле, но BSS (в отличие от сегмента данных) не занимает фактического места в объектном файле.

Еще раз: BSS - это старая директива ассемблера - Block Started by Symbol - которую можно игнорировать; сегодня сегмент BSS - это сегмент неинициализированных данных VAS процесса.

Теперь мы видим, почему файл `a.out` разбит на сегменты. Сегменты удобно отображать в объекты, которые компоновщик среды выполнения может загружать напрямую. Загрузчик берет каждый образ сегмента в файле и помещает его в память. Сегменты становятся областями памяти исполняемой программы, каждый из которых имеет определенное назначение. Это показано на рисунке



How the Segments of an Executable are Laid Out in Memory

Текстовый сегмент содержит инструкции программы. Загрузчик копирует это прямо из файла в память (обычно с системным вызовом `mmap()`), и больше не нужно беспокоиться об этом, поскольку текст программы обычно никогда не меняется ни по значению, ни по размеру. Некоторые операционные системы и компоновщики могут даже назначать соответствующие разрешения для различных разделов в сегментах, например, текст можно сделать доступным только для чтения и выполнения, некоторые данные можно сделать доступными только для чтения и записи, другие данные можно сделать доступными только для чтения и т. д.

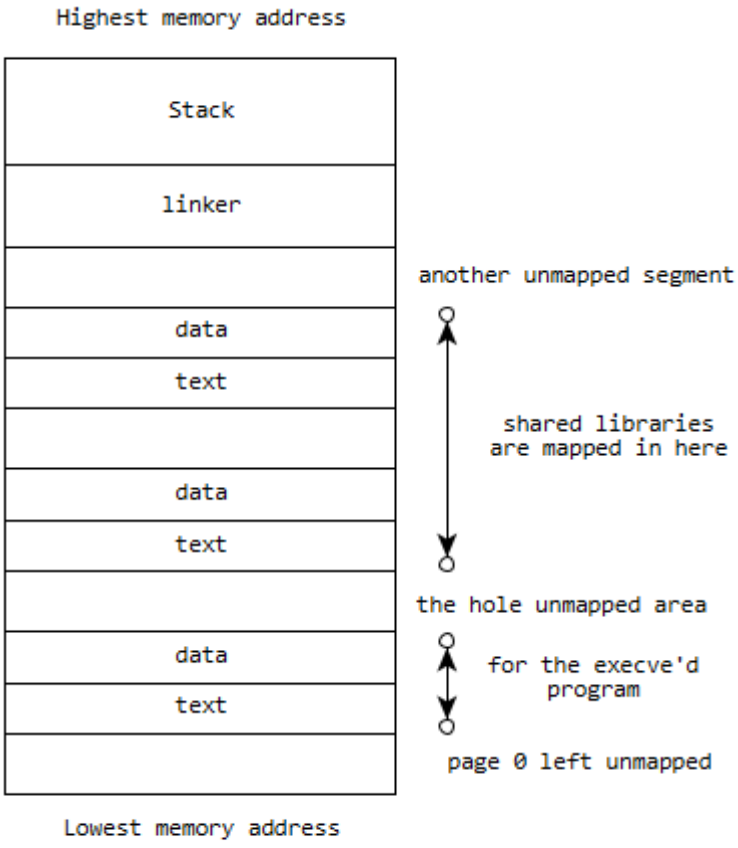
Сегмент данных содержит инициализированные глобальные и статические переменные с присвоенными им значениями. Размер сегмента BSS затем получается из исполняемого файла, и загрузчик получает блок этого размера, помещая его сразу после сегмента данных. Этот блок обнуляется, когда он помещается в адресное пространство программы. На этом этапе весь объем данных и BSS обычно называют сегментом данных. Это связано с тем, что сегмент в терминах управления памятью ОС представляет собой просто диапазон последовательных виртуальных адресов, поэтому смежные сегменты объединяются. Сегмент данных обычно

является самым большим сегментом в любом процессе.

На рисунке показана структура памяти программы перед самым началом ее выполнения. Нужно некоторое пространство в памяти для локальных переменных, временных файлов, передачи параметров при вызовах функций и т. д. Для этого выделяется сегмент стека. Также нужно место в куче для динамически выделяемой памяти. Все это будет выделено по требованию, как только будет сделан первый вызов `malloc()`.

Обратите внимание, что нижняя часть виртуального адресного пространства не отображается; то есть оно находится в адресном пространстве процесса, но не назначено физическому адресу, поэтому любые ссылки на него будут ошибочными. Обычно это несколько килобайт памяти с нулевого адреса.

При использовании разделяемых библиотек, адресное пространство процесса будет выглядеть примерно так



Virtual Address Space Layout, Showing Shared Libraries

Теперь об организации структур данных выполняющейся программы. Существует ряд структур данных среды выполнения: стек, записи активации, данные, куча и так далее. Далее рассмотрим каждый из них по очереди и проанализируем поддерживаемые ими функции Си.

Сегмент стека

Сегмент стека содержит единственную структуру данных - стек. Стек - это динамическая

область памяти, которая реализует очередь «последним пришел - первым ушел». Классическое определение стека гласит, что на нем может быть любое количество данных, но единственные допустимые операции - это добавить или удалить данные из верхней части стека. Операция `push` увеличивает стек, а pop удаляет из него значение. Разработчики компиляторов придерживаются более гибкого подхода. Добавление или удаляе данных - только сверху, но также можно изменять значения, которые находятся в середине стека. Функция может обращаться к переменным, локальным для вызывающей функции, через параметры или глобальные указатели. Среда выполнения поддерживает указатель в регистре `sp`, который указывает текущую вершину стека. Сегмент стека имеет три основных применения, два из которых связаны с функциями и одно с вычислением выражений:

• Стек предоставляет место для хранения локальных переменных, объявленных внутри функций. В терминологии Си они известны как «автоматические переменные».

• Стек хранит "служебную" информацию, используемую при вызове функции. Эта служебная информация известна как кадр стека или, в более общем смысле, запись активации процедуры. Он включает адрес, с которого была вызвана функция (то есть, куда вернуться, когда вызываемая функция завершена), любые параметры, которые не подходят в регистры и сохраненные значения регистров.

• Стек также работает как блокнот - каждый раз, когда программе требуется какое-то временное хранилище, например, для вычисления длинного арифметического выражения, она может помещать частичные результаты в стек, выталкивая их при необходимости. Хранилище, полученное вызовом `alloca()`, также находится в стеке. Не используйте `alloca()` для получения памяти, содержимое которой вы хотите сохраненить после выхода из функции, которая ее выделяет. (Она будет перезаписана при следующем вызове функции.)

Стек был бы не нужен, если бы не рекурсивные вызовы. Если бы не они, фиксированный объем пространства для локальных переменных, параметров и адресов возврата был бы известен во время компиляции и мог бы быть выделен в BSS. Ранние реализации BASIC, COBOL и FORTRAN не допускали рекурсивных вызовов функций, поэтому им не требовался динамический стек во время выполнения. Разрешение рекурсивных вызовов означает, что мы должны найти способ разрешить одновременное существование нескольких экземпляров локальных переменных, хотя будут доступны только самые последние созданные - классическая спецификация стека.

Задача 2.1.

Напишите программу, чтобы узнать, когда закончится `time_t`.

Подсказка:

1. Посмотрите определение `time_t`. Это находится в файле `/usr/include/time.h`.
2. Создайте программу, чтобы поместить наибольшее значение в переменную типа time_t, затем передайте его в `ctime()`, чтобы преобразовать в строку ASCII. Распечатайте строку. Обратите внимание, что `ctime` не имеет ничего общего с языком C, он просто означает «конвертировать время».

Задача 2.2.

Посмотрите на сегменты в исполняемом файле.

1. Скомпилируйте программу `"hello world"`, запустите `ls -l` для исполняемого файла, чтобы получить его общий размер, и запустите size, чтобы получить размеры сегментов внутри него.

2. Добавьте объявление глобального массива из 1000 int, перекомпилируйте и повторите измерения. Обратите внимание на различия.

3. Теперь добавьте начальное значение в объявление массива (помните, C не заставляет вас указывать значение для каждого элемента массива в инициализаторе). Это переместит массив из сегмента BSS в сегмент данных. Повторите измерения. Обратите внимание на различия.

4. Теперь добавьте объявление большого массива в локальную функцию. Объявите второй большой локальный массив с инициализатором. Повторите измерения. Данные определены локально внутри функции, хранящейся в исполняемом файле? Какая разница, инициализирован он или нет?

5. Какие изменения происходят с размерами файлов и сегментов, если вы компилируете для отладки? Для максимальной оптимизации?

Проанализируйте результаты , чтобы убедиться, что:

- сегмент данных хранится в объектном файле
- сегмент BSS не сохраняется в объектном файле (за исключением примечания к его требованиям к размеру времени выполнения)
- текстовый сегмент больше всего подвержен оптимизации
- На размер файла `a.out` влияет компиляция для отладки, но не на сегменты.

Задача 2.3.

Скомпилируйте и запустите тестовую программу, чтобы определить примерное расположение стека в вашей системе:

```
#include <stdio.h>
int main() {
    int i;
    printf("The stack top is near %p\n", &i);
    return 0;
}
```

Найдите расположение сегментов данных и текста, а также кучу внутри сегмента данных, объявив переменные, которые будут помещены в эти сегменты, и распечатав их адреса. Увеличьте размер стека, вызвав функцию и объявив несколько больших локальных массивов. Какой сейчас адрес вершины стека?

Примечание: стек может располагаться по разным адресам на разных архитектурах и для разных версий ОС. Хотя мы говорим о вершине стека, на большинстве процессоров стек растет вниз, к адресам памяти с меньшими значениями.

Задача 2.4.

Ваша задача исследовать стек процесса или вспомнить то, как это делается. Вы можете

"заглянуть" в стек процессов (стек main()) разными способами:

- Автоматически через утилиту gstack (1)
- Вручную с помощью отладчика GDB

Пользователи Ubuntu здесь могут столкнуться с проблемой. На момент написания (Ubuntu 18.04) gstack, похоже, не был доступен для Ubuntu (и его альтернатива, pstack, тоже может не работать). Пожалуйста, используйте второй метод (через GDB), как показано ниже.

Сначала посмотрите на стек при помощи gstack(1). В качестве примера ниже приведен стек bash (параметр команды - PID процесса):

```
$ gstack 14654
#0 0x00007f3539ece7ea in waitpid () from /lib64/libc.so.6
#1 0x000056474b4b41d9 in waitchld.isra ()
#2 0x000056474b4b595d in wait_for ()
#3 0x000056474b4a5033 in execute_command_internal ()
#4 0x000056474b4a52c2 in execute_command ()
#5 0x000056474b48f252 in reader_loop ()
#6 0x000056474b48dd32 in main ()
$
```

Номер кадра стека появляется слева перед символом #. Обратите внимание, что кадр #0 - это самый нижний кадр. Читайте стек снизу вверх, то есть от кадра #6 (кадра для функции main()) до кадра #0 (кадра для функции waitpid()). Обратите внимание, что если процесс многопоточный, gstack покажет стек каждого потока.

Теперь загляните в стек User-режима через GDB. Чтобы заглянуть в стек процессов, используем GDB для присоединения к процессу.

Ниже приведена небольшая тестовая программа на языке C, которая выполняет несколько вызовов вложенных функций. Граф вызовов будет выглядеть следующим образом:

```
main() --> foo() --> bar() --> bar_is_now_closed() --> pause()
```

Системный вызов `pause(2)` - пример блокирующего вызова. Он переводит вызывающий процесс в спящий режим, ожидая (или блокируя) событие; событие, которое здесь блокируется, - это доставка любого сигнала процессу.

Код программы `stacker.c`:

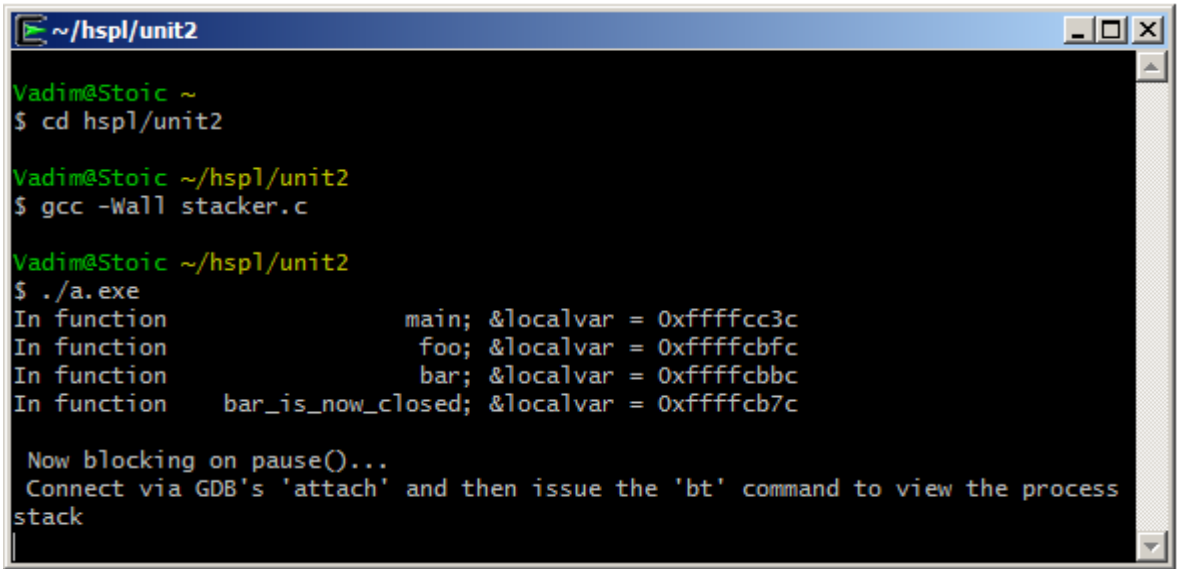
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

#define MSG "In function %20s; &localvar = %p\n"

static void bar_is_now_closed(void) {
    int localvar = 5;
    printf(MSG, __FUNCTION__, &localvar);
    printf("\n Now blocking on pause()...\n"
```

```
        " Connect via GDB's 'attach' and then issue the 'bt' command"
        " to view the process stack\n");
    pause();
}
static void bar(void) {
    int localvar = 5;
    printf(MSG, __FUNCTION__, &localvar);
    bar_is_now_closed();
}
static void foo(void) {
    int localvar = 5;
    printf(MSG, __FUNCTION__, &localvar);
    bar();
}

int main(int argc, char **argv) {
    int localvar = 5;
    printf(MSG, __FUNCTION__, &localvar);
    foo();
    exit (EXIT_SUCCESS);
}
```



Теперь откройте GDB, в нем подключитесь (`attach`) к процессу (в примере ниже это PID 24957) и исследуйте стек с помощью команды backtrace (bt):

```
$ gdb --quiet
(gdb) attach 24957
Attaching to process 24957
Reading symbols from <...>/hspl/unit2/stacker...done.
Reading symbols from /lib64/libc.so.6...Reading symbols from
/usr/lib/debug/usr/lib64/libc-2.26.so.debug...done.
done.
Reading symbols from /lib64/ld-linux-x86-64.so.2...Reading symbols
...
(gdb) bt
...
```

Hint: В Ubuntu из-за безопасности GDB не позволяет подключаться к какому-либо процессу; это можно обойти, запустив GDB от имени пользователя root.

А теперь тот же процесс через gstack:

```
$ gstack 24957
...
```

gstack - это на самом деле «wrapper shell script», который вызывает неинтерактивно GDB и выдает команду backtrace, которую вы только что использовали. В качестве дальнейшего упражнения ознакомьтесь с выводом gstack.

Задача 2.5.

Известно, что при вызове процедур и возврате из них процессор использует стек. Можно ли в такой схеме обойтись без счетчика команд (IP), используя вместо него вершину стека? Свой ответ обоснуйте и покажите пример(ы).