基础知识

Linux 进程内存布局

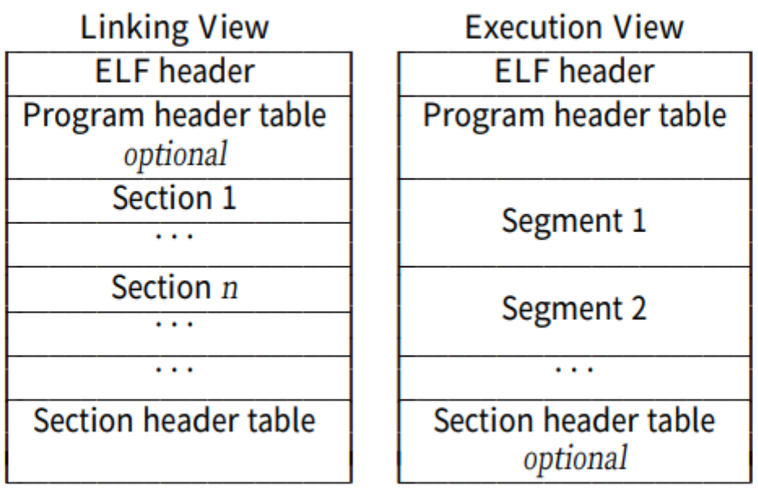
ELF 文件布局

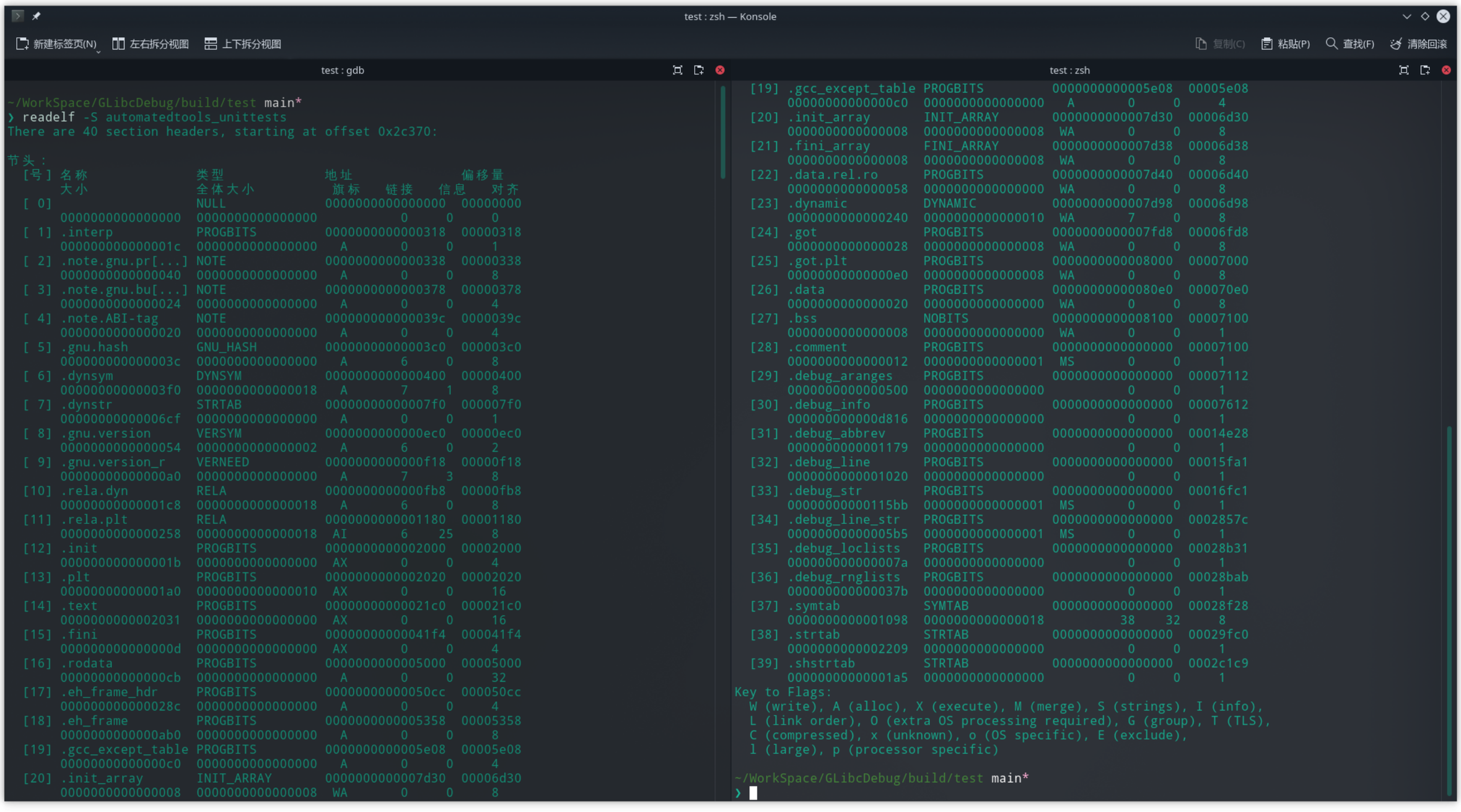
- 参考文章-Linux 可执行文件如何装载进虚拟内存
- 参考文章-ELF 文件解析（一）：Segment 和 Section
- 参考文章-Executable and Linkable Format (ELF)

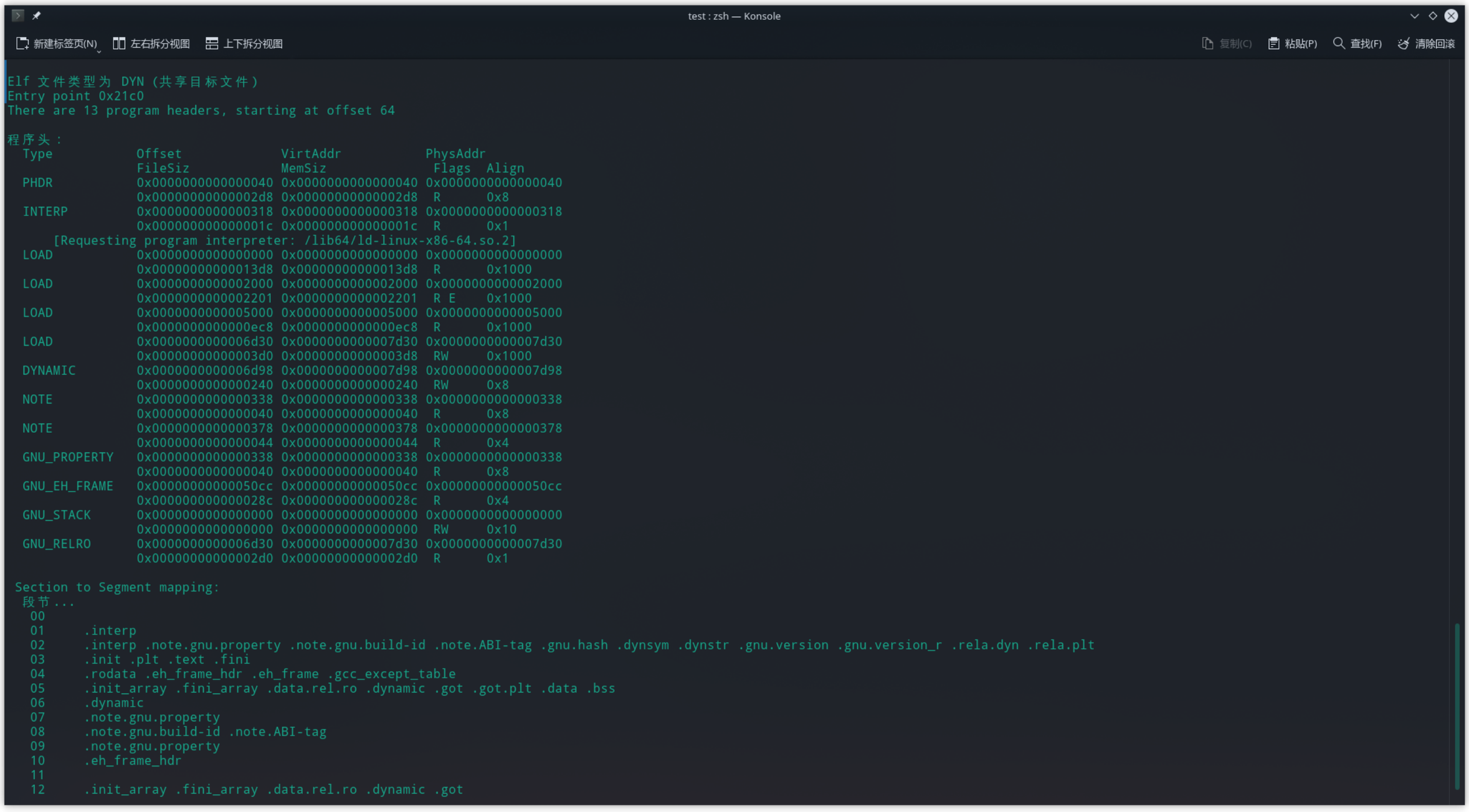
Linux ELF 格式可执行程序的结构是以段（segment）为单位组合而成的。Linux 系统在装载 elf 程序文件时，通过 loader 将各个 segment 依次载入到某一地址开始的空间中：

> section 代指 .text、.bss、.data 等，这些信息可用于链接时地址重定位等，因此 section 又被称为链接视图，可以使用 readelf -S object 来查看目标文件的 section。
>
> segment 则是执行视图，程序执行是以 segment 为单位载入，每个 segment 对应 ELF 文件中 program header table 中的一个条目，用来建立可执行文件的进程映像。通常说的，代码段、数据段是 segment，目标代码中的 section 会被链接器组织到可执行文件的各个 segment 中。可以使用 readelf -l a.out 查看 ELF 格式执行文件的 segment 信息。

Figure 1-1: Object File Format







如上图所示，LOAD 的段是运行时所需载入的：

```
LOAD           0x0000000000002000 0x0000000000002000 0x0000000000002000
               0x0000000000002201 0x0000000000002201  R E    0x1000
# 其标号为 0x3，对应：

03     .init .plt .text .fini

# 可以发现其包含我们的代码段，R（只读）E（可执行）W（可写）
```

ELF 文件装载

- 参考文章-ELF 文件的加载过程
- 参考文章-Linux 中 main 是如何执行的

Linux 支持不同的可执行程序，其内核使用 `struct linux_binfmt` 来描述各种可执行程序，以 ELF 格式举例：

1. 首先需要填充一个 linux_binfmt 格式的结构：

```c
// /fs/binfmt.c
static struct linux_binfmt elf_format = {
    .module       = THIS_MODULE,
    .load_binary = load_elf_binary, //通过读存放在可执行文件中的信息为当前进程建立一个新的执行环境
    .load_shlib       = load_elf_library,  //用于动态的把一个共享库捆绑到一个已经在运行的进程，这是由 uselib() 系统调用激活的
    .core_dump        = elf_core_dump,
    .min_coredump     = ELF_EXEC_PAGESIZE,
    .hasvdso      = 1
};
```

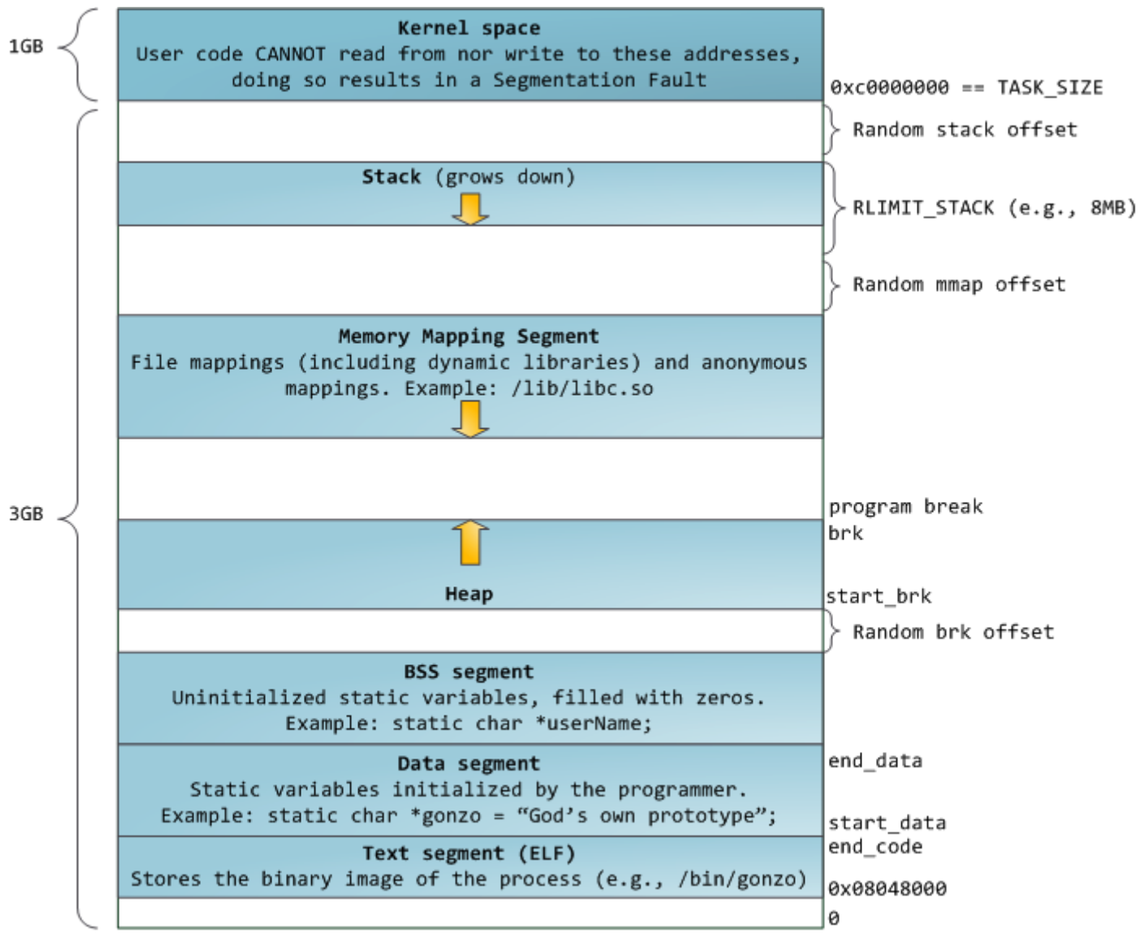
2. 所有的 `linux_binfmt` 对象都处于一个链表中，第一个元素的地址存放在 formats 变量中，可以通过调用 register_binfmt() 和 unregister_binfmt() 函数在链表中插入和删除元素，在系统启动期间，为每个编译进内核的可执行格式都执行 registre_fmt() 函数。执行一个可执行程序的时候，内核会 `list_for_each_entry` 遍历所有注册的 `linux_binfmt` 对象，对其调用 `load_binrary` 方法来尝试加载，直到加载成功为止
3. 内核通过 `execv()` 或 `execve()` 系统调用到 `do_execve()` 函数，其读取 ELF 文件头部字节（Linux 128 字节头），再通过 `search_binary_handler()` 搜索对应的格式加载方法，最终调用处理函数来加载目标 ELF。
4. 以 ELF 格式的 load_elf_binary 函数执行过程举例：
    - 填充并且检查目标程序 ELF 头部
    - load_elf_phdrs 加载目标程序的程序头表
    - 如果需要动态链接，则寻找和处理解释器段
    - 检查并读取解释器的程序表头
    - 装入目标程序的段 segment
    - 填写程序的入口地址
    - create_elf_tables 填写目标文件的参数环境变量等必要信息
    - start_kernel 宏准备进入新的程序入口

进程内存布局

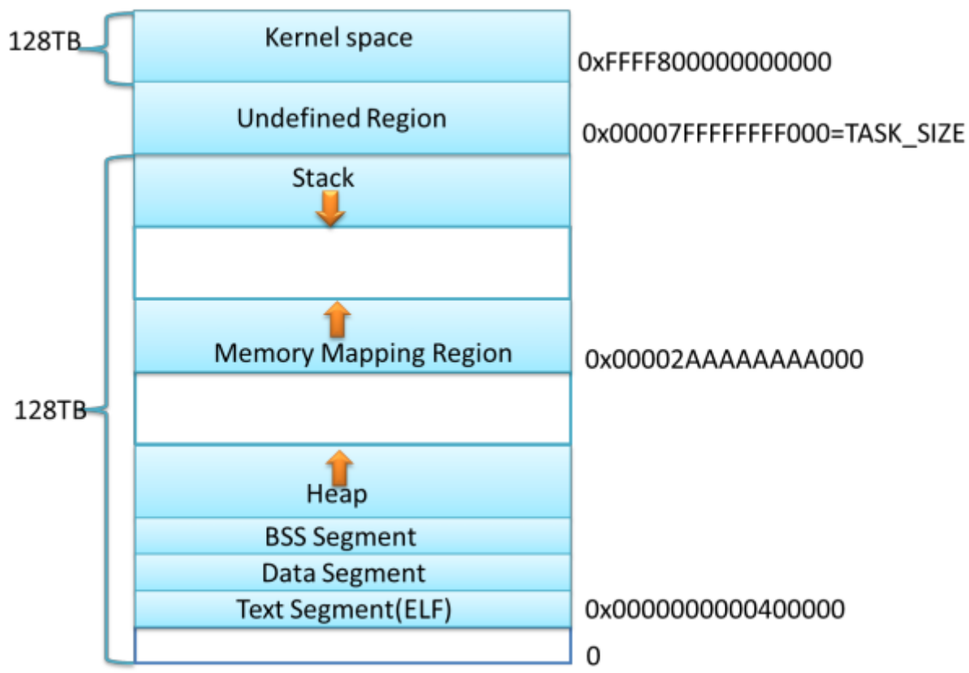
- 参考文章-进程的内存空间布局
- 参考文章-anatomy-of-program-in-memory

1. 内核态内存空间，大小固定（编译时可以调整），32 位系统一般为 1GB，64 位系统为 128TB。
2. 用户态的栈，大小不固定，可以用 ulimit -s 进行调整，默认一般为 8M，从高地址向低地址增长。
3. mmap 区域（内存映射段），既可以从高地址到低地址延伸（所谓 flexible layout），也可以从低到高延伸（所谓 legacy layout)，看进程具体情况。一般 32 位系统默认为 flexible，64 位系统为 legacy。
4. brk 区域（堆），紧邻数据段（甚至贴着），从低位向高位伸展，但它的大小主要取决于 mmap 如何增长，一般来说，即使是 32 位的进程以传统方式延伸，也有差不多 1 GB 的空间。
5. 数据段，主要是进程里初始化和未初始化（BSS）的全局数据总和，当然还有编译器生成一些辅助数据结构等等），大小取决于具体进程，其位置紧贴着代码段。
6. 代码段，主要是进程的指令，包括用户代码和编译器生成的辅助代码，其大小取决于具体程序，但起始位置根据 32 位还是 64 位一般固定（-fPIC, -fPIE 等除外）。
7. 以上各段（除了代码段数据段）其起始位置根据系统是否起用 randomize_va_space 一般稍有变化，各段之间因此可能有随机大小的间隔。

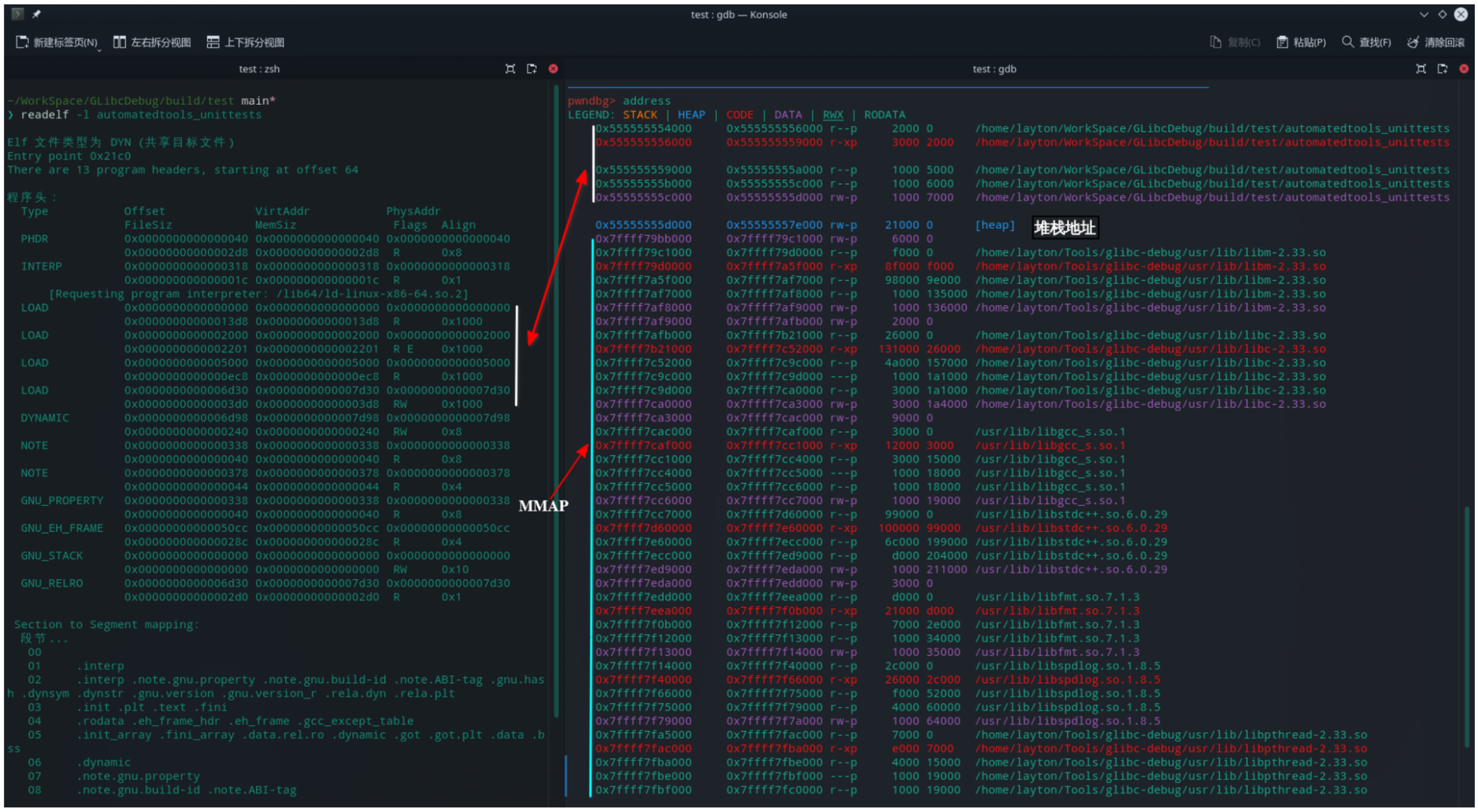
- 32 位 flexible layout。

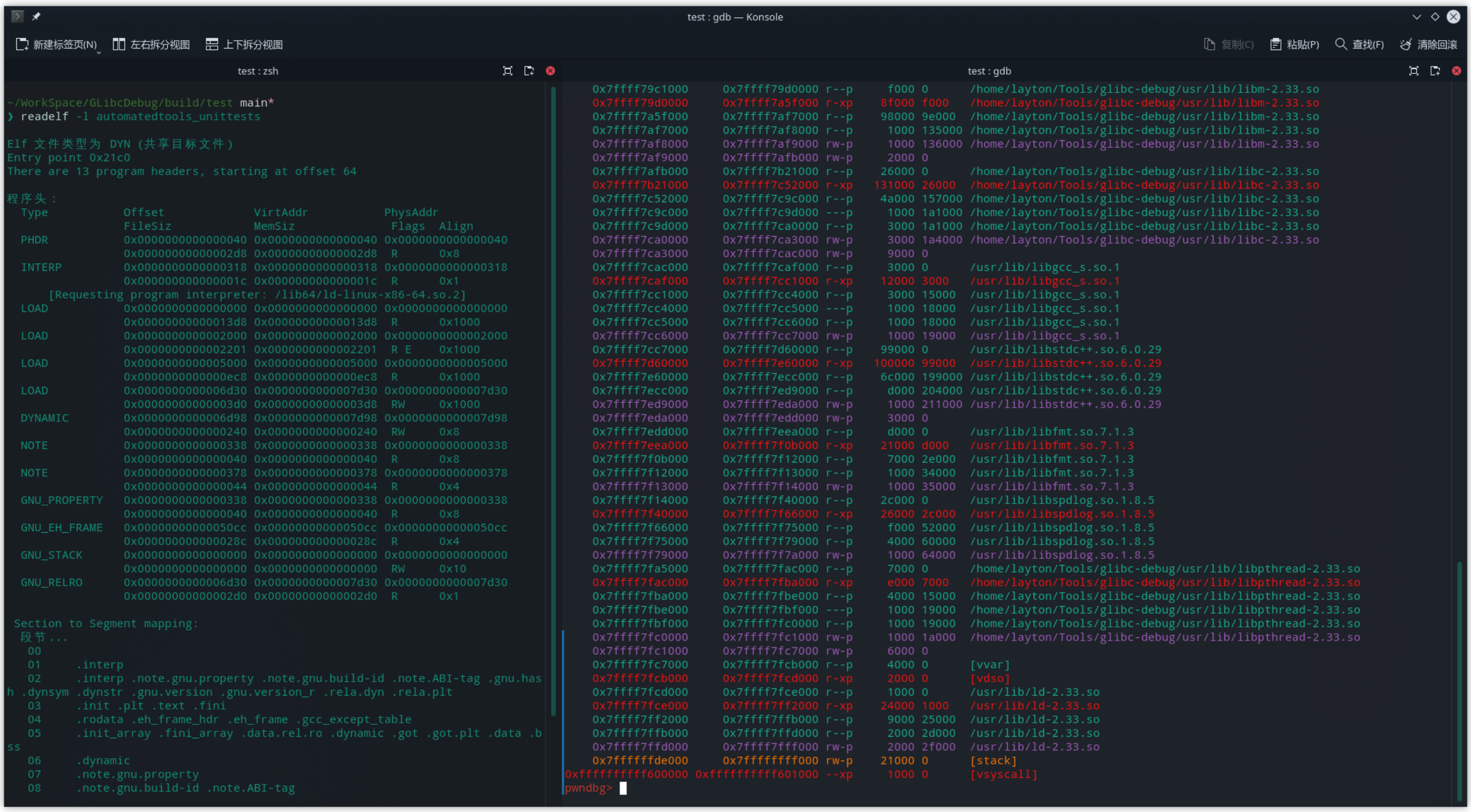


- 64 位 legacy layout。



- 查看 segment 实际加载到内存中的地址情况 64 位系统：





操作系统内存分配的相关函数

Heap 和 mmap 区域都可以供用户自由使用，但是它在刚开始的时候并没有映射到内存空间内，是不可访问的。在向内核请求分配该空间之前，对这个空间的访问会导致 segmentation fault。用户程序可以直接使用系统调用来管理 heap 和 mmap 映射区域，但更多的时候程序都是使用 C 语言提供的 malloc() 和 free() 函数来动态的分配和释放内存。

堆

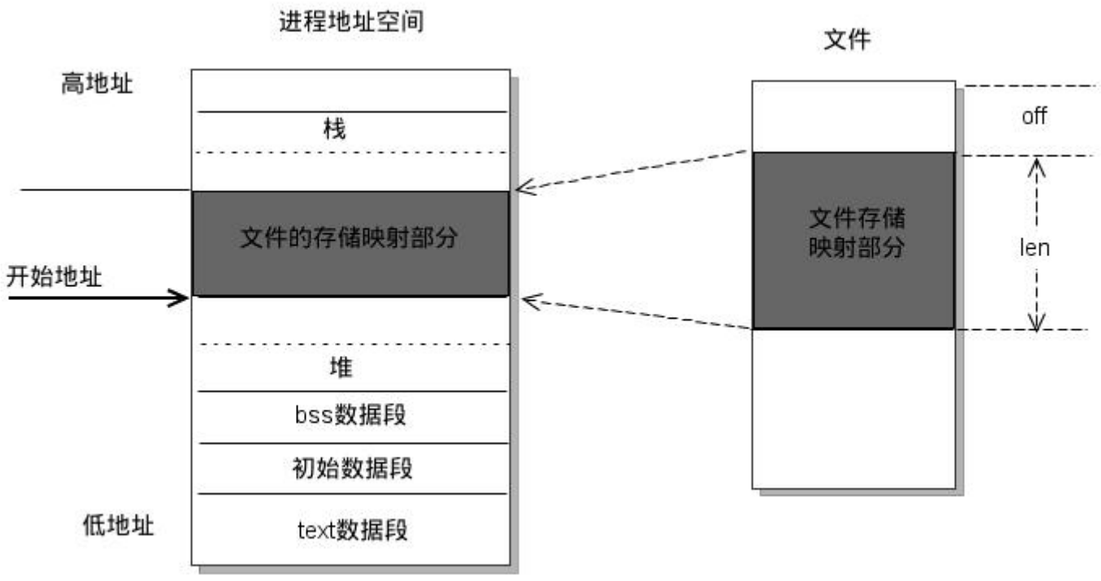
- 参考文章-Glibc：浅谈 malloc() 函数具体实现

Linux 操作系统对于堆操作提供了 `brk` 函数，基于 brk 函数 glibc 封装了 `sbrk` ，同时 malloc 函数族包括 realloc、calloc 等最终都是调用 sbrk() 函数将数据段下移，并在内核的管理下将虚拟地址空间映射到内存供 malloc 函数使用。

mmap 映射区

- 参考文章-认真分析 mmap

mmap 映射区的操作主要通过，mmap() 函数。即一种内存映射文件的方法，其将一个文件或者其他对象映射到进程内存 mmap 分区，实现文件磁盘地址与进程虚拟地址空间中一段虚拟内存映射的关系，使得进程可以通过访问虚拟内存操作对应的文件或数据，系统会自动通过回写脏页到对应的文件磁盘。



常见内存管理的方法

C 风格的内存管理

主要通过 malloc 与 free 函数，主要通过 brk 或者 mmap 来对进程添加额外的虚拟内存。

- 优点：灵活、适应性强、速度快。
- 缺点：对于大块的内存释放，可能由于再等待其中某一小块内存而被缓存。内存的生存周期也不容易管理。

池式内存管理

内存池是一种半内存管理方法。内存池帮助某些程序进行自动内存管理，这些程序会经历一些特定的阶段，而且每个阶段中都有分配给进程的特定阶段的内存。

- 优点：应用程序可以简单地管理内存。内存分配和回收更快，因为每次都是在一个池中完成的。可以预先分配错误处理池。有非常易于使用的标准实现。
- 缺点：内存池只适用于操作可以分阶段的程序。不能与三方库很好的合作。如果程序的结构发生变化需要修改内存池。需要用户管理内存。

引用计数

在引用计数中，所有共享的数据结构都有一个域来包含当前活动“引用”结构的次数。当计数清 0 释放内存。

- 优点：实现简单。易于使用。
- 缺点：需要多余缓存引用结构。异常处理不友好。引用结构存放位置影响 CPU 读取数据。

垃圾回收

全自动地检测并移除不再使用的数据对象。垃圾收集器通常会在当可用内存减少到少于一个具体的阈值时运行。以程序的栈、全局变量、寄存器出发追踪这些数据指向的每一块内存，而没有被这些数据指向的则会被回收。

- 优点：简单、易用，不宜出错。
- 缺点：效率低，用户没法指定释放内存，出错后调试困难，如果没有将内存引用置为 NULL，依旧会有内存泄漏的风险。

内存管理器的设计目标

1. 兼容性。
2. 可移植性。
3. 内存管理消耗的内存要小。
4. 速度要快。
5. 便于调试。
6. 最大化局部性：尽量保持访问内存的连续性，便于 CPU Cache 缓存。
7. 方便的参数配置。
8. 默认配置的适应性要好。

Ptmalloc 内存管理概述

Ptmalloc 对比老版本 malloc 支持了多线程的支持。分配置处在用户程序和内核之间，它响应用户的分配请求，向操作系统申请内存，然后将其返回给用户程序，为了保持高效的分配，分配器一般都会预先分配一块大于用户请求的内存，并通过某种算法管理这块内存。来满足用户的内存分配要求，用户释放掉的内存也并不是立即就返回给操作系统，相反，分配器会管理这些被释放掉的空闲空间，以应对用户以后的内存分配要求。

Ptmalloc 的设计假设

- 参考文章-系统调用与内存管理

1. 具有长生命周期的大内存分配使用 mmap。
2. 特别大的内存分配总是使用 mmap.
3. 具有短生命周期的内存分配使用 sbrk。
4. 尽量只缓存临时使用的空闲小内存块，当大内存块或者是生命周期较长的大内存块在释放时都直接归还给操作系统。
5. 需要长期存储的程序不合适用 ptmalloc 来管理。
6. 对空闲的小内存块只会在 malloc 和 free 的时候进行合并。free 时空闲内存块可能放入内存池中，而不是立即归还给操作系统。
7. 为了支持多线程，多个线程可以同一个分配区中分配内存，ptmalloc 假设线程 A 释放掉一块内存后，线程 B 申请类似大小的内存，但是 A 释放的内存跟 B 需要的内存不一定完全相等。