

1、尝试使用time系统调用统计一段时间内程序指令的执行情况

现在实现：ptrace跟踪程序执行，并且每过一段时间延时采样一次

2、如何实现软件切分？

设想：根据采样的数据，统计访问频率较多的页。然后保存原始ELF文件对应逻辑地址的页数据，将其他数据切除达到切分效果。

TODO:

1.看内核是如何将ELF文件映射到地址空间中的

fs/binfmt_elf.c

所有的 `linux_binfmt` 对象都处于一个单向链表中，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 `register_binfmt()` 函数。当实现了一个新的可执行格式的模块正被装载时，也执行这个函数，当模块被卸载时，执行 `unregister_binfmt()` 函数。

```
static int __init init_elf_binfmt(void)
{
    return register_binfmt(&elf_format);
}

static void __exit exit_elf_binfmt(void)
{
    /* Remove the COFF and ELF loaders. */
    unregister_binfmt(&elf_format);
}
```

Linux 允许用户注册自己定义的可执行格式。对这种格式的识别或者通过存放在文件前 128 字节的魔数，或者通过表示文件类型的扩展名。例如，MS-DOS 的扩展名由 “.” 把三个字符从文件名中分离出来：`.exe` 扩展名标识可执行文件，而 `.bat` 扩展名标识 shell 脚本。

当内核确定可执行文件是自定义格式时，它就启动相应的解释程序(*interpreter program*)。解释程序运行在用户态，读入可执行文件的路径名作为参数，并执行计算。例如，包含 Java 程序的可执行文件就由 Java 虚拟机（如 `/usr/lib/java/bin/java`）来解释。

`sys_execve()` 把可执行文件路径名拷贝到一个新分配的页框。然后调用 `do_execve()` 函数，传递给它的参数为指向这个页框的指针、指针数组的指针及把用户态寄存器内容保存到内核态堆栈的位置。`do_execve()` 依次执行下列操作：

1. 动态地分配一个 `linux_binprm` 数据结构，并用新的可执行文件的数据填充这个结构。

2. 调用 `path_lookup()`、`dentry_open()` 和 `path_release()`，以获得与可执行文件相关的目录项对象、文件对象和索引节点对象。如果失败，则返回相应的错误码。
3. 检查是否可以由当前进程执行该文件，再检查索引节点的 `i_writcount` 字段，以确定可执行文件没被写入；把 -1 存放在这个字段以禁止进一步的写访问。
4. 在多处理器系统中，调用 `sched_exec()` 函数来确定最小负载 CPU 以执行新程序，并把当前进程转移过去（参见第七章）。
5. 调用 `init_new_context()` 检查当前进程是否使用自定义局部描述符表，参见第二章“Linux LDT”一节）。如果是，函数为新程序分配和准备一个新的 LDT。
6. 调用 `prepare_binprm()` 函数填充 `linux_binprm` 数据结构，这个函数又依次执行下列操作：
 - a. 再一次检查文件是否可执行（至少设置一个执行访问权限）。如果不可执行，则返回错误码（因为带有 `CAP_DAC_OVERRIDE` 权能的进程总能通过检查，所以第 3 步中的检查还不够。参见本章前面“进程的信任状和权能”一节）。
 - b. 初始化 `linux_binprm` 结构的 `e_uid` 和 `e_gid` 字段，考虑可执行文件的 `setuid` 和 `setgid` 标志的值。这些字段分别表示有效的用户 ID 和组 ID。也要检查进程的权能（在本章前面的“进程的信任状和权能”一节中介绍了兼容性技巧）。
 - c. 用可执行文件的前 128 字节填充 `linux_binprm` 结构的 `buf` 字段。这些字节包含的是适合于识别可执行文件格式的一个魔数和其他信息。

```
static void fill_elf_header(struct elfhdr *elf, int segs,
                           u16 machine, u32 flags, u8 osabi)
{
    memset(elf, 0, sizeof(*elf));

    memcpy(elf->e_ident, ELFMAG, SELFMAG);
    elf->e_ident[EI_CLASS] = ELF_CLASS;
    elf->e_ident[EI_DATA] = ELF_DATA;
    elf->e_ident[EI_VERSION] = EV_CURRENT;
    elf->e_ident[EI_OSABI] = ELF_OSABI;

    elf->e_type = ET_CORE;
    elf->e_machine = machine;
    elf->e_version = EV_CURRENT;
    elf->e_phoff = sizeof(struct elfhdr);
    elf->e_flags = flags;
    elf->e_ehsize = sizeof(struct elfhdr);
    elf->e_phentsize = sizeof(struct elf_phdr);
    elf->e_phnum = segs;

    return;
}

static int load_elf_binary(struct linux_binprm *bprm, struct pt_regs *regs);
static int load_elf_library(struct file *);
static unsigned long elf_map(struct file *, unsigned long, struct elf_phdr *,
                             int, int, unsigned long);
```

include/linux/binfmts.h:

```
struct linux_binprm{
    char buf[BINPRM_BUF_SIZE];
#ifdef CONFIG_MMU
    struct vm_area_struct *vma;
#else
    # define MAX_ARG_PAGES 32
    struct page *page[MAX_ARG_PAGES];
#endif
    struct mm_struct *mm;
    unsigned long p; /* current top of mem */
    unsigned int
        cred_prepared:1, /* true if creds already prepared (multiple
                           * preps happen for interpreters) */
        cap_effective:1; /* true if has elevated effective capabilities,
                           * false if not; except for init which inherits
                           * its parent's caps anyway */
#ifdef __alpha__
    unsigned int taso:1;
#endif
    unsigned int recursion_depth;
    struct file * file;
    struct cred *cred; /* new credentials */
    int unsafe; /* how unsafe this exec is (mask of LSM_UNSAFE_*) :
    unsigned int per_clear; /* bits to clear in current->personality */
    int argc, envc;
    char * filename; /* Name of binary as seen by procs */
    char * interp; /* Name of the binary really executed. Most
                   of the time same as filename, but could be
                   different for binfmt_{misc,script} */

    unsigned interp_flags;
    unsigned interp_data;
    unsigned long loader, exec;
};
```

加载二进制形式（文件）的linux_binfmt结构体

```
/*
 * This structure defines the functions that are used to load the binary formats th
at
 * linux accepts.
 */
struct linux_binfmt {
    struct list_head lh;
    struct module *module;
    int (*load_binary)(struct linux_binprm *, struct pt_regs * regs);
    int (*load_shlib)(struct file *);
    int (*core_dump)(struct coredump_params *cprm);
    unsigned long min_coredump; /* minimal dump size */
    int hasvdso;
};
```

fs/binfmt_elf.c中有其具体实现:

```
static struct linux_binfmt elf_format = {  
    .module      = THIS_MODULE,  
    .load_binary = load_elf_binary,  
    .load_shlib  = load_elf_library,  
    .core_dump   = elf_core_dump,  
    .min_coredump = ELF_EXEC_PAGESIZE,  
    .hasvdso     = 1  
};
```