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_writecount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 procps */
        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;
};
```

linux_binprm和linux_binfmt的区别：

在 Linux 内核中，`linux_binprm` 和 `linux_binfmt` 是两个相关但不同的概念。

1. **`linux_binprm` 结构体：**
   - `linux_binprm` 结构体用于表示执行一个二进制程序所需的参数和状态。它包含了执行可执行文件时的一些关键信息，例如文件名、参数、环境变量、文件标识符、进程的地址空间等。
   - 在执行可执行文件时，内核会创建一个 `linux_binprm` 结构体，用于保存执行过程中的状态和信息。这个结构体在执行的各个阶段都会被使用，以确保正确加载和执行可执行文件。
2. **`linux_binfmt` 结构体：**
   - `linux_binfmt` 是用于表示二进制格式（binary format）的结构体。它用于注册和管理在 Linux 内核中执行不同二进制格式的处理程序。
   - 在 Linux 内核中，不同的二进制格式（如 ELF、a.out 等）有不同的处理程序，用于解释和执行这些格式的可执行文件。`linux_binfmt` 结构体定义了这些处理程序所需的函数指针，包括加载可执行文件、执行可执行文件等。
   - 一个典型的 `linux_binfmt` 结构体包含了如下函数指针：
     - `load_binary`：加载可执行文件的函数。
     - `load_shlib`：加载共享库的函数。
     - `core_dump`：生成核心转储文件的函数等。

在 Linux 2.6.34 内核中，`linux_binprm` 和 `linux_binfmt` 通常在可执行文件加载和执行的过程中协同工作。`linux_binprm` 结构体用于保存执行时的状态，而 `linux_binfmt` 结构体则定义了与不同二进制格式相关的处理程序。在执行过程中，内核通过注册的 `linux_binfmt` 结构体找到合适的处理程序，并使用 `linux_binprm` 结构体保存执行过程中的状态。

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
        };
```

include/linux/elf.h

#define elfhdr elf32_hdr or #define elfhdr elf64_hdr

elfhdr的具体定义：

```
//LLD elf32_hdr define -> elfhdr define
typedef struct elf32_hdr{
  unsigned char e_ident[EI_NIDENT];
  Elf32_Half    e_type;
  Elf32_Half    e_machine;
  Elf32_Word    e_version;
  Elf32_Addr    e_entry;  /* Entry point */
  Elf32_Off     e_phoff;
  Elf32_Off     e_shoff;
  Elf32_Word    e_flags;
  Elf32_Half    e_ehsize;
  Elf32_Half    e_phentsize;
  Elf32_Half    e_phnum;
  Elf32_Half    e_shentsize;
  Elf32_Half    e_shnum;
  Elf32_Half    e_shstrndx;
} Elf32_Ehdr;
```

```
/* 32位 ELF 头部结构体 */
struct elf32_hdr {
    unsigned char e_ident[EI_NIDENT]; /* ELF标识 */
    Elf32_Half    e_type;             /* 对象文件类型 */
    Elf32_Half    e_machine;          /* 架构类型 */
    Elf32_Word    e_version;          /* ELF版本 */
    Elf32_Addr    e_entry;            /* 程序入口点的虚拟地址 */
    Elf32_Off     e_phoff;            /* 程序头表在文件中的偏移量 */
    Elf32_Off     e_shoff;            /* 节头表在文件中的偏移量 */
    Elf32_Word    e_flags;            /* 处理器相关标志 */
    Elf32_Half    e_ehsize;           /* ELF头部的大小 */
    Elf32_Half    e_phentsize;        /* 程序头表中一个入口的大小 */
    Elf32_Half    e_phnum;            /* 程序头表中入口的数量 */
    Elf32_Half    e_shentsize;        /* 节头表中一个入口的大小 */
    Elf32_Half    e_shnum;            /* 节头表中入口的数量 */
    Elf32_Half    e_shstrndx;         /* 节名字符串表的索引 */
};
```