操作系统 Lab 04 内存页编程

姓名: 雍崔扬
学号: 21307140051

Task 1

1.1 概述

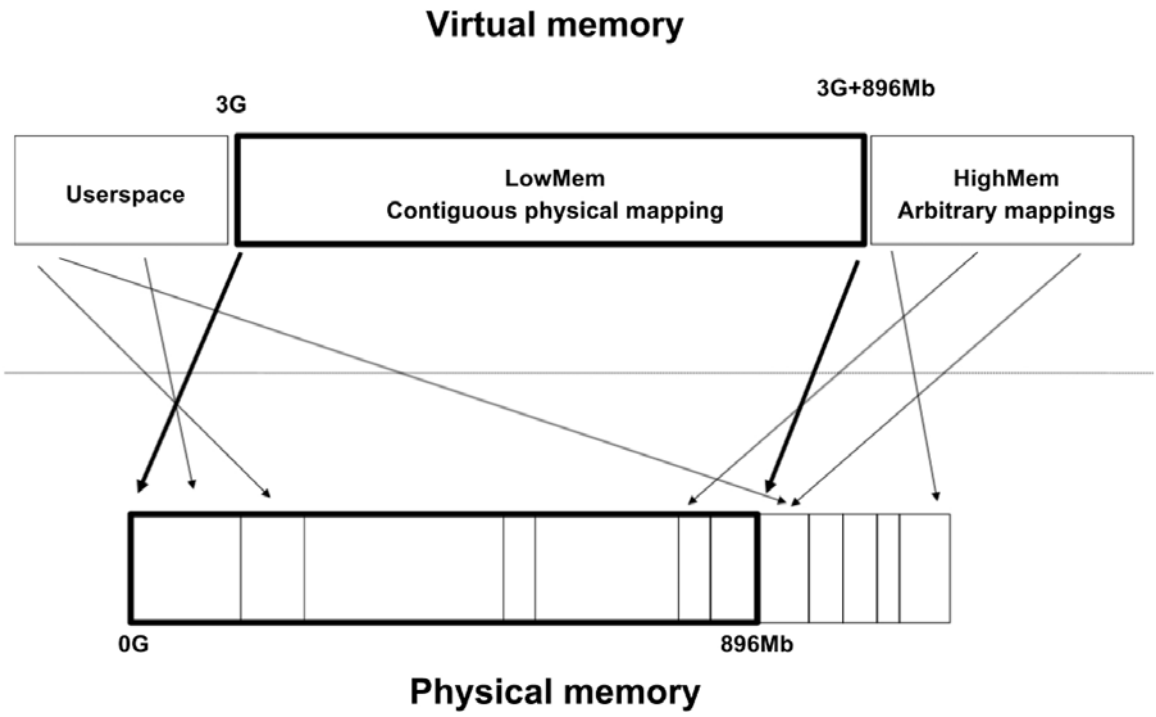
在 Linux 内核中，可以将内核地址空间映射到用户地址空间.
这种机制能够消除在用户空间和内核空间之间来回拷贝数据的开销.
我们可以通过设备驱动程序以及用户空间的设备接口 (如 `/dev`) 来实现这种映射.

如何实现这个功能呢?
通过在设备驱动程序的 `struct file_operations` 中实现 `mmap()` 系统调用，就可以实现这种映射功能.
虚拟内存管理的基本单位是页面，页面的大小通常为 `4K`，但在某些平台上可以达到 `64K`.
在使用虚拟内存时，涉及两种类型的地址: 虚拟地址和物理地址.
所有的 CPU 访问 (包括来自内核空间的访问) 都使用虚拟地址，
通过 MMU (内存管理单元) 和页表将虚拟地址转换为物理地址.

一个物理页面由页帧号 (PFN) 标识.
PFN 可以通过将物理地址除以页面大小来计算 (或通过将物理地址向右移动 PAGE_SHIFT 位来获得)

为了提高效率，虚拟地址空间被划分为用户空间和内核空间.
同样出于效率的原因，内核空间包含一个称为 `lowmem` 的内存映射区域，
该区域在物理内存中连续映射，从最低的物理地址 (通常为 `0`) 开始.
`PAGE_OFFSET` 宏则定义了 `lowmem` 被映射的虚拟地址.

在 32 位系统中，并非所有可用的内存都可以映射到 `lowmem`.
因此内核空间中设有一个称为 `highmem` 的独立区域，可以用来任意映射物理内存.
通过 `kmalloc()` 分配的内存在 `lowmem` 中，且是物理连续的;
而通过 `vmalloc()` 分配的内存则不是连续的，并且不在 `lowmem` 中 (它有一个位于 `highmem` 的专用区域)



1.2 用于内存映射的结构体

在讨论设备内存映射机制之前，我们将介绍 Linux 内存管理子系统中使用的一些基本结构体:

- ① `struct page` 用于包含系统中所有物理页面的信息
  内核为物理内存中的每一个页面都维护一个 `struct page` 结构体.
  (注意 `page` 结构体是用来描述物理内存页的，而不是内存映射的虚拟页)
  有许多函数与该结构体交互:
  - `virt_to_page`: 返回与虚拟地址关联的页面
  - `pfn_to_page`: 返回与页帧号关联的页面
  - `page_to_pfn`: 返回与 `struct page` 关联的页帧号
  - `page_address`: 返回 `struct page` 的虚拟地址. 此函数只能用于低端内存 (`lowmem`) 的页面
  - `kmap`: 在内核中为任意物理页面 (可来自高端内存 `highmem`) 创建映射，并返回一个可用于直接引用页面的虚拟地址
- ② `struct vm_area_struct` 保存有关连续虚拟内存区域的信息.
  可以通过 `procfs` 查看进程的内存区域，具体可在 `/proc/[PID]/maps` 文件中找到，例如:

```
root@qemux86:~# cat /proc/1/maps
# address           perms offset   device inode      pathname
08048000-08050000 r-xp 00000000   fe:00 761        /sbin/init.sysvinit
08050000-08051000 r--p 00007000   fe:00 761        /sbin/init.sysvinit
08051000-08052000 rw-p 00008000   fe:00 761        /sbin/init.sysvinit
092e1000-09302000 rw-p 00000000   00:00 0          [heap]
4480c000-4482e000 r-xp 00000000   fe:00 576        /lib/ld-2.25.so
4482e000-4482f000 r--p 00021000   fe:00 576        /lib/ld-2.25.so
4482f000-44830000 rw-p 00022000   fe:00 576        /lib/ld-2.25.so
44832000-449a9000 r-xp 00000000   fe:00 581        /lib/libc-2.25.so
449a9000-449ab000 r--p 00176000   fe:00 581        /lib/libc-2.25.so
449ab000-449ac000 rw-p 00178000   fe:00 581        /lib/libc-2.25.so
449ac000-449af000 rw-p 00000000   00:00 0
b7761000-b7763000 rw-p 00000000   00:00 0
```

```
b7763000-b7766000 r--p 00000000   00:00 0          [vvar]
b7766000-b7767000 r-xp 00000000   00:00 0          [vdso]
bfa15000-bfa36000 rw-p 00000000   00:00 0          [stack]
```

示例显示了每个内存区域的地址范围、权限、偏移、设备和关联文件.
一个内存区域的基本特征包括起始地址、结束地址、长度和权限.
每次从用户空间调用 `mmap()` 时，都会创建一个新的内存区域并关联到一个 `struct vm_area_struct` 结构体.
驱动程序若支持 `mmap()` 操作，也必须初始化相应的 `struct vm_area_struct`

此结构体的关键字段包括：

- `vm_start` 和 `vm_end`：内存区域的起始和结束地址
- `vm_file`：指向关联文件的指针 (如果有的话)
- `vm_pgoff`：文件中区域的偏移
- `vm_flags`：包含该区域的权限和特性标志
- `vm_ops`：操作函数集合，用于该区域的自定义行为
- `vm_next` 和 `vm_prev`：链接同一进程中相邻的内存区域，形成一个链表结构

- ③ `struct mm_struct` 包含与进程关联的所有内存区域.
  每个进程的 `struct task_struct` 中的 `mm` 字段指向该进程的 `struct mm_struct` 结构体
  通过 `/proc/<pid>/maps` 文件，可以查看与特定进程相关的内存区域映射.

1.3 设备驱动程序内存映射

内存映射是 Unix 系统中的一个重要特性.
对于驱动程序而言，内存映射功能允许用户空间设备直接访问内存.
要在驱动程序中支持内存映射，需要在设备驱动程序的 `struct file_operations` 结构体中实现 `mmap` 字段.

当实现了 `mmap` 字段后，用户空间进程可以对与设备关联的文件描述符调用 `mmap()` 系统调用.
`mmap()` 的函数原型如下：

```
void *mmap(caddr_t addr, size_t len, int prot, int flags, int fd, off_t offset);
```

在设备和用户空间之间映射内存时，用户进程需要先打开设备，然后使用所得的文件描述符调用 `mmap()` 系统调用.

设备驱动程序的 `mmap()` 函数定义如下:

```
int (*mmap)(struct file *filp, struct vm_area_struct *vma);
```

- `filp` 字段是一个指向 `struct file` 的指针，表示用户空间打开设备时创建的文件结构.
- `vma` 字段用于指示设备应将内存映射到的虚拟地址空间.

在实现驱动程序的 `mmap()` 操作时，通常会先分配内存 (如通过 `kmalloc()`、`vmalloc()`、`alloc_pages()`),
然后使用辅助函数 (如 `remap_pfn_range()`) 将物理内存映射到用户空间的虚拟地址.
`remap_pfn_range()` 函数会将连续的物理地址空间映射到由 `vma` 参数指定的虚拟地址空间中，其函数定义如下:

```
int remap_pfn_range(struct vm_area_struct *vma, unsigned long addr,
                    unsigned long pfn, unsigned long size, pgprot_t prot);
```

- `vma`: 用于映射的虚拟内存空间
- `addr`: 映射开始的虚拟地址
- `pfn`: 需要映射到的页帧号
- `size`: 映射内存的大小 (以字节为单位)
- `prot`: 此映射的保护标志，用于设置访问权限

以下代码展示了如何使用 `remap_pfn_range()` 函数将从页帧号 `pfn` 开始的物理内存 (先前已分配) 连续映射到虚拟地址 `vma->vm_start`，直到 `vma->vm_start + len`:

```
struct vm_area_struct *vma;
unsigned long len = vma->vm_end - vma->vm_start;
int ret;

ret = remap_pfn_range(vma, vma->vm_start, pfn, len, vma->vm_page_prot);
if (ret < 0) {
    pr_err("could not map the address area\n");
    return -EIO;
}
```

如何获取页帧号 `pfn` 取决于内存的分配方式.
不同分配方式的 `pfn` 获取方法如下:

- **使用 `kmalloc()` 分配的内存:**

  ```
  static char *kmalloc_area;
  unsigned long pfn = virt_to_phys((void *)kmalloc_area) >> PAGE_SHIFT;
  ```

- **使用 `vmalloc()` 分配的内存:**

  ```
  static char *vmalloc_area;
  unsigned long pfn = vmalloc_to_pfn(vmalloc_area);
  ```

  `vmalloc()` 分配的内存不是物理连续的，因此映射该内存范围时需要逐页映射，并为每一页计算物理地址.
  `vmalloc()` 适用于需要大块空间的情况，因为它可以将多个不连续的物理页拼接成一个连续的虚拟页.

- **使用 `alloc_pages()` 分配的内存:**

  ```
  struct page *page;
  unsigned long pfn = page_to_pfn(page);
  ```

由于这些页面被映射到用户空间，它们可能会被交换出.
为了避免这种情况，必须设置页面的 `PG_reserved` 位.
可以使用 `SetPageReserved()` 设置该位，并在释放内存前使用 `ClearPageReserved()` 重置该位:

```
void alloc_mmap_pages(int npages)
{
    int i;
    char *mem = kmalloc(PAGE_SIZE * npages);
    if (!mem)
        return mem;
    for(i = 0; i < npages * PAGE_SIZE; i += PAGE_SIZE)
        SetPageReserved(virt_to_page(((unsigned long)mem) + i));
```

```
    return mem;
}

void free_mmap_pages(void *mem, int npages)
{
    int i;
    for(i = 0; i < npages * PAGE_SIZE; i += PAGE_SIZE)
        ClearPageReserved(virt_to_page(((unsigned long)mem) + i));
    kfree(mem);
}
```

这些函数分别在分配和释放内存时设置和清除 `PG_reserved` 位，以确保这些页面在映射到用户空间时不会被交换出.

1.4 实验任务

由于 Linux 操作系统的内存管理机制，**内核空间**和**用户空间**是彼此隔离的.
即便内核分配了内存，用户空间的程序仍然无法直接访问它，除非显式地将其映射到用户空间.
这就是 `mmap` 的作用.

(1) 映射连续的物理内存到用户空间

在 `linux/tools/labs` 路径下生成代码框架:

```
LABS=memory_mapping/test make skels
LABS=memory_mapping/kmmap make skels
```

任务描述:
实现一个设备驱动程序，将连续的物理内存 (例如通过 `kmalloc()` 获得的内存) 映射到用户空间.
请根据已学知识生成任务代码，填写标记为 **TODO 1** 的部分.

提示:

- ① **分配内存:** 在模块初始化函数中，使用 `kmalloc()` 分配大小为 `NPAGES+2` 页的内存区域.
  为确保内存地址对齐，找到该区域中第一个页边界对齐的地址.
  - 页大小: `PAGE_SIZE`
  - 将分配的内存存储在 `*kmalloc_ptr` 中，将页边界对齐的地址存储在 `kmalloc_area` 中
  - 对齐计算: 使用 `PAGE_ALIGN()` 确定 `kmalloc_area` 地址
- ② **设置 `PG_reserved` 位:** 在每个页上启用 `PG_reserved` 位
  - 使用 `SetPageReserved()` 设置每页的 `PG_reserved` 位
  - 在释放内存前，用 `ClearPageReserved()` 清除该位
  - 页转换: 使用 `virt_to_page()` 将虚拟页地址转换为对应的物理页结构体
- ③ **初始化页内容:** 为测试程序验证，每页的前 4 个字节需填入特定值 `0xaa, 0xbb, 0xcc, 0xdd`
- ④ **实现 `mmap()` 函数:** 在驱动函数 `mmap()` 中执行映射操作
  - 使用 `remap_pfn_range()` 实现映射操作
  - `remap_pfn_range()` 的第三个参数是页帧号 (PFN),
    需将虚拟地址转换为物理地址，然后右移 `PAGE_SHIFT` 位以获得 PFN.

实现: (位于 `linux/tools/labs/skels/memory_mapping/kmmap/kmmap.c` 中)

- **(1) `my_init` 函数:**

- ① 使用 `kmalloc()` 分配大小为 `NPAGES+2` 页的内存区域 (确保对齐后的区域足够大容纳 `NPAGES` 页的数据):

  ```
  kmalloc_ptr = kmalloc((NPAGES + 2) * PAGE_SIZE, GFP_KERNEL);
  ```

  `GFP_KERNEL` 表示以内核模式分配内存，因此可以发生睡眠.

- ② 使用 `PAGE_ALIGN()` 进行页边界对齐:

  ```
  kmalloc_area = (char *)PAGE_ALIGN((unsigned long)kmalloc_ptr)
  ```

  `PAGE_ALIGN()` 是一个宏，用于屏蔽掉地址中的低位，其实现通常是这样的:

  ```
  #define PAGE_ALIGN(addr) ((addr) & ~(PAGE_SIZE - 1))
  ```

- ③ 设置每页的 `PG_reserved` 位以避免页交换:

  ```
  for (i = 0; i < NPAGES; i++) {
      SetPageReserved(virt_to_page(kmalloc_area + i * PAGE_SIZE));
  }
  ```

- ④ 将每页的前 4 个字节填入特定值 `0xaa, 0xbb, 0xcc, 0xdd`

  ```
  for (i = 0; i < NPAGES; i++) {
      char *page_start = kmalloc_area + i * PAGE_SIZE;
      page_start[0] = 0xaa;
      page_start[1] = 0xbb;
      page_start[2] = 0xcc;
      page_start[3] = 0xdd;
  }
  ```

- **(2) `my_mmap` 函数:**
  将分配的连续物理内存映射到用户空间:

  ```
  ret = remap_pfn_range(vma, vma->vm_start,
                        virt_to_phys((void *)kmalloc_area) >> PAGE_SHIFT,
                        length, vma->vm_page_prot);
  ```

  - `vma`: 指向用户空间的虚拟内存区域 (`vm_area_struct` 结构体) 的指针
  - `vma->vm_start`: 用户空间的虚拟地址起点
  - `virt_to_phys((void *)kmalloc_area) >> PAGE_SHIFT`: 起始页面的物理页帧号
  - `length`: 要映射的字节数
  - `vma->vm_page_prot`: 页面保护属性 (如读写权限)，继承自用户虚拟内存区域

- **(3) `my_exit` 函数:**
  清空 `PG_reserved` 字段并释放 `kmalloc` 申请的物理空间:

  ```
  for (i = 0; i < NPAGES; i++) {
      ClearPageReserved(virt_to_page(kmalloc_area + i * PAGE_SIZE));
  }
  kfree(kmalloc_ptr);
  ```

测试:
在内核模块编译完成后，使用 `make build` 进行编译 (如果遇到返回值缺少错误，就给对应的函数补充 `return 0`)
加载模块后，运行框架自带的测试程序以验证映射:

```
qemux86 login: root
root@qemux86:~# cd skels/memory_mapping/kmmap
root@qemux86:~/skels/memory_mapping/kmmap# insmod kmmap.ko
root@qemux86:~/skels/memory_mapping/kmmap# lsmod | grep kmmap
kmmap 16384 0 - Live 0xe0839000 (O)
root@qemux86:~/skels/memory_mapping/kmmap# cd ..
root@qemux86:~/skels/memory_mapping# ./test/mmap-test 1
matched
matched
matched
matched
matched
matched
matched
matched
matched
matched
matched
matched
matched
matched
matched
matched
root@qemux86:~/skels/memory_mapping#
```

(2) 映射非连续的物理内存到用户空间

在 `linux/tools/labs` 路径下生成代码框架:

```
LABS=memory_mapping/vmmap make skels
```

任务描述:
实现一个设备驱动程序，将非连续的物理内存 (例如通过 `vmalloc()` 获得的内存) 映射到用户空间.
在生成任务代码后，填写标记为 `TODO 1` 的部分.

提示:

1. 使用 `vmalloc()` 分配一个大小为 `NPAGES` 的内存区域，并将分配的区域存储在 `*vmalloc_area` 中.
   `vmalloc()` 分配的内存是页对齐的.
2. 在每个页上启用 `PG_reserved` 位，以防止这些页被系统交换出内存.
   使用 `SetPageReserved()` 设置该位.
   在释放内存之前，记得用 `ClearPageReserved()` 清除该位.
3. 使用 `vmalloc_to_page()` 将虚拟页转换为物理页，
   以便使用 `SetPageReserved()` 和 `ClearPageReserved()`.
4. 为了使用后面的测试验证程序功能，请在每个页的前 4 个字节中填入以下值: `0xaa`, `0xbb`, `0xcc`, `0xdd`.

5. 实现 `mmap` 驱动函数:
    - 使用 `vmalloc_to_pfn()` 将虚拟的 `vmalloc` 地址转换为物理地址.
      `vmalloc_to_pfn()` 函数直接返回一个 `PFN` (页帧号)
    - 注意 `vmalloc` 分配的页在物理内存中不是连续的,
      因此需要对每个页使用 `remap_pfn_range()` 进行映射.
    - 遍历所有虚拟页，确定每个页的物理地址，并使用 `remap_pfn_range()` 进行映射.
      请确保每次都确定物理地址，并使用一个页的范围进行映射.

实现: (位于 `linux/tools/labs/skels/memory_mapping/vmmap/vmmap.c` 中)

- **(1) `my_init` 函数:**
    - ① 使用 `vmalloc()` 分配 `NPAGES` 个页面:

      ```
      vmalloc_area = vmalloc(NPAGES * PAGE_SIZE);
      ```

      `vmalloc` 返回的内存在物理上是分散的 (即不保证物理连续)，但在虚拟地址空间中是连续的.
    - ② 设置每页的 `PG_reserved` 位以避免页交换:

      ```
      for (i = 0; i < NPAGES; i++) {
          SetPageReserved(vmalloc_to_page(vmalloc_area + i * PAGE_SIZE));
      }
      ```

    - ③ 将每页的前 4 个字节填入特定值 `0xaa, 0xbb, 0xcc, 0xdd`

      ```
      for (i = 0; i < NPAGES; i++) {
          char *page_ptr = vmalloc_area + i * PAGE_SIZE;
          page_ptr[0] = 0xaa;
          page_ptr[1] = 0xbb;
          page_ptr[2] = 0xcc;
          page_ptr[3] = 0xdd;
      }
      ```

- **(2) `my_mmap` 函数:**
  遍历所有虚拟页，确定每个页的物理地址，并使用 `remap_pfn_range()` 进行映射:

  ```
  for (i = 0; i < NPAGES; i++) {
      pfn = vmalloc_to_pfn(vmalloc_area_ptr);
      ret = remap_pfn_range(vma, start, pfn, PAGE_SIZE, vma->vm_page_prot);
      start += PAGE_SIZE;
      vmalloc_area_ptr += PAGE_SIZE;
  }
  ```

    - `vma`: 指向用户空间的虚拟内存区域 (`vm_area_struct` 结构体) 的指针
    - `start`: 用户空间的虚拟地址起点
    - `pfn`: 当前页面的物理页帧号
    - `PAGE_SIZE`: 要映射的页面大小
    - `vma->vm_page_prot`: 页面保护属性 (如读写权限)，继承自用户虚拟内存区域
- **(3) `my_exit` 函数:**
  清空 `PG_reserved` 字段并释放 `vmalloc` 申请的物理空间:

```
for (i = 0; i < NPAGES; i++) {
    ClearPageReserved(vmalloc_to_page(vmalloc_area + i * PAGE_SIZE));
}
vfree(vmalloc_area);
```

测试:
在内核模块编译完成后，使用 `make build` 进行编译 (如果遇到返回值缺少错误，就给对应的函数补充 `return 0`)
加载模块后，运行框架自带的测试程序以验证映射:

```
qemux86 login: root
root@qemux86:~# cd skels/memory_mapping/vmmap
root@qemux86:~/skels/memory_mapping/vmmap# insmod vmmap.ko
vmmap: loading out-of-tree module taints kernel.
root@qemux86:~/skels/memory_mapping/vmmap# lsmod | grep vmmap
vmmap 16384 0 - Live 0xe0876000 (O)
root@qemux86:~/skels/memory_mapping/vmmap# cd ..
root@qemux86:~/skels/memory_mapping# ./test/mmap-test 1
matched
matched
matched
matched
matched
matched
matched
matched
matched
matched
matched
matched
matched
matched
matched
matched
matched
root@qemux86:~/skels/memory_mapping# rmmod vmmap.ko
root@qemux86:~/skels/memory_mapping# lsmod | grep vmmap
root@qemux86:~/skels/memory_mapping#
```

(3) 思考题

- 为什么在映射内存到用户空间时，需要设置 `PG_reserved` 位?
  答: 在内核中将 `PG_reserved` 位设置为 `1` 是为了确保该页面在内核内存管理过程中不会被交换或释放
  从而保证映射的一致性.
- 什么是虚拟内存，为什么需要使用虚拟内存，和物理内存有什么区别?
  答: 物理内存是实际硬件提供的内存容量，
  而虚拟内存是操作系统为进程提供的抽象，使每个进程都可以认为自己拥有一个完整连续的内存空间.
  虚拟内存的作用包括:
  - **隔离进程:** 每个进程有独立的虚拟地址空间，避免彼此干扰，增强安全性

- **扩展内存容量:** 通过交换机制将不常用的数据暂时存储在硬盘上，使程序可以使用超过物理内存容量的空间
- **简化内存管理:** 虚拟地址可以通过内存映射技术灵活地映射到不同的物理内存区域

Task 2

本实验通过实现一个内核模块，旨在掌握如何在内核空间分配可执行页面，
并使用字符设备驱动与 `ioctl` 接口进行用户态和内核态的交互.
实验中，内核模块为用户态函数提供可执行页面并在内核中执行该函数，
以此深入探讨了 Just-In-Time (JIT) 编译的实现原理及其安全性问题.

2.1 字符设备驱动与 `ioctl` 的实现

在该模块中，我们通过实现一个字符设备驱动来提供接口，
使用户态程序能够向内核模块传递函数信息和执行指令，完成用户态和内核态的交互.
实现细节包括字符设备的注册和 `ioctl` 接口的设计.

- **(1) 字符设备驱动:**
  字符设备驱动通过以下 API 进行注册和卸载，供用户态与内核模块通信:
  - 使用 `register_chrdev(0, "JIT", &fops)` 注册字符设备，命名为 "JIT"
  - 使用 `unregister_chrdev(0, "JIT")` 卸载字符设备

  注册成功后，用户态程序可以通过该字符设备访问内核模块，进行函数传输和调用.
- **(2) `ioctl` 接口:**
  `ioctl` (输入输出控制) 用于在用户态和内核态之间传递控制指令.
  该模块的 `ioctl` 实现两个主要功能:
  - 接收用户态函数的地址和长度信息，将其移入内核空间的可执行页面.
  - 执行加载到可执行页面中的函数.

  用户态程序通过 `ioctl` 接口发起指令，传递包含函数信息的结构体参数给内核模块.
  以下是 `ioctl` 传递参数的示例代码片段，与实验代码类似:

```c
struct user_data {
    void *func_addr;
    size_t func_len;
};

// 用户态中传递函数信息
struct user_data data = { .func_addr = user_function, .func_len = func_size
};
int fd = open("/dev/JIT", O_RDWR);
ioctl(fd, CMD_LOAD_FUNC, &data);
```

  在内核模块中，通过 `copy_from_user` 函数从用户态读取数据，以确保数据传输的安全性:

```c
struct user_data kernel_data;
if (copy_from_user(&kernel_data, user_arg, sizeof(struct user_data)) != 0) {
    return -EFAULT; // 数据传输失败
}
```

在该例子中，`copy_from_user` 函数将用户态的 `user_data` 结构体内容复制到内核态的 `kernel_data` 结构体中.
之所以不能直接使用 `memcpy`，是因为用户进程和内核进程不共享页表.
用户地址在内核模块中是一个无效地址，无法直接访问，
因此需要借助 `copy_from_user` 等内核提供的接口来安全地读取用户态数据.

2.2 JIT 及其安全性

Just-In-Time (JIT) 编译器是一种动态编译技术，
它可以在代码执行过程中将代码即时编译为机器码，从而提升执行效率.
然而 JIT 编译在内核中的应用引入了执行权限管理和安全性问题，
特别是在设置可执行页面的读、写、执行 (RWX) 权限方面.

为了缓解 RWX 页面的安全风险，现代系统通常采用临时权限切换机制.
该机制通过动态调整页面权限来增强安全性，例如:

- 从可写 (RW) 切换到可执行 (RX)，在执行后再切换回可写 (RW)
- 将不可执行 (NX) 页面临时设置为可写，以确保 JIT 编译过程中安全地加载代码

这些措施帮助减少潜在的安全隐患，相关的实现和详细信息可以参考 Linux 安全文档.

2.3 在内核中更改页权限

在内核模块中，可以通过 `set_memory_rw()`、`set_memory_x()` 和 `set_memory_ro()` 函数来修改页面的读、写和执行权限.
以下示例代码展示了这些权限更改的基本用法:

```
// 将页面设置为只读
set_memory_ro((unsigned long)page_addr, 1);

// 将页面设置为可执行
set_memory_x((unsigned long)page_addr, 1);
```

常见的权限更改流程:

- **读写到只读再到可执行:**
  通常先将页面设置为只读 (RO)，然后再切换为可执行 (X)
- **写入到执行的切换:**
  当需要写入时，先将页面设置为不可执行 (NX)，允许写操作 (RW)
  完成写入后，再根据需要调整权限.

注意: 在较新版本的内核中，这些权限调整函数已不再对内核模块开放.
本实验的助教提供了特殊方法，使这些函数可以在模块中使用，便于实验过程中对页面权限的动态调整.

2.4 实验任务

将分发的代码框架 `kernelJIT` 目录复制到 `skels` 目录下.
可以创建一个名为 `lab4` 的文件夹以便管理实验文件.

(1) 代码阅读与修改

阅读 `kernelJIT.c` 和 `testJIT.c` 文件源码，理解代码框架.
注意其中标明为 "Black Magic" 的部分不要求理解.
根据前述的字符设备驱动与权限调整内容，完善 `kernelJIT.c` 文件中的 `TODO` 部分.

`kernelJIT.c` 文件中:

- ① `JIT_init` **函数:**

```
static int __init JIT_init(void) {
    fn_kallsyms_lookup_name_init();

    device_major = register_chrdev(0, "JIT", &fops);
    if (device_major < 0) {
        printk(KERN_ERR "JIT: failed to register char device\n");
        return device_major;
    }

    /* TODO: allocate a page with kmalloc */
    // 使用 kmalloc() 分配一页内存，并将返回的地址赋值给 exec_page
    // GFP_KERNEL 表示在内核空间进行分配 (可睡眠)
    exec_page = (unsigned long)kmalloc(PAGE_SIZE, GFP_KERNEL);

    if (!exec_page) {
        return -ENOMEM;
    } else {
        printk(KERN_INFO "JIT: allocated executable page at %lx\n",
exec_page);
    }

    kernel_function = (int (*)(void))exec_page;

    printk(KERN_INFO "JIT: module loaded\n");
    return 0;
}
```

- ② `JIT_ioctl` **函数:**

```
static long JIT_ioctl(struct file *filp, unsigned int cmd, unsigned long
arg) {
    struct user_function uf;

    switch (cmd) {
        case JIT_IOCTL_LOAD_FUNC: {
            /* TODO: set memory permissions (make it writable) */
            // 在执行代码拷贝之前，我们需要先确保目标内存页 (exec_page) 是可写的
            // 因为我们将从用户空间拷贝函数代码到这个内存页
            set_memory_rw(exec_page, 1);

            /* TODO: copy in the uf structure */
            // 使用 copy_from_user() 将用户传递的 user_function 结构体中的数据
            // (包括函数地址 addr 和长度 length) 拷贝到内核空间的 uf 结构体中
            if (copy_from_user(&uf, (void __user *)arg, sizeof(struct
user_function))) {
                return -EFAULT;
            }
```

```
            /* TODO: copy the function code to exec_page */
            // 使用 memcpy() 将用户空间的函数代码从 uf.addr 拷贝到 exec_page 中
            memcpy((void *)exec_page, (void *)uf.addr, uf.length);

            return 0;
        }
        case JIT_IOCTL_EXECUTE_FUNC: {
            /* TODO: set memory permissions (make it executable) */
            // 将 exec_page 所在的内存页的权限设置为可执行
            set_memory_x(exec_page, 1);

            /* call kernel_function */
            return kernel_function(); // Executes the function at exec_page
        }
        default:
            return -EINVAL;
    }

    return 0;
}
```

- ③ `JIT_exit` **函数:**

```
static void __exit JIT_exit(void) {
    unregister_chrdev(device_major, "JIT");

    /* TODO: set memory permissions (back to rw) */
    set_memory_rw(exec_page, 1); // Set the page back to read-write
permissions

    /* TODO: free the page */
    kfree((void *)exec_page); // Free the allocated page

    printk(KERN_INFO "JIT: module exited\n");
}
```

`testJIT.c` **文件中:**

```
int main()
{
    int fd;
    struct user_function uf;

    fd = open("/dev/JIT", O_RDWR);
    if (fd < 0) {
        perror("open");
        return -1;
    }

    uf.addr = my_user_function;
    uf.length = (size_t)my_user_function_end - (size_t)my_user_function; /*
Calculate the length of the function */

    /* TODO: Add ioctl to register the function */
    // 将 uf 结构体传递给内核模块，告诉内核要加载哪个用户空间的函数
```

```
    ioctl(fd, JIT_IOCTL_LOAD_FUNC, &uf);

    /* TODO: Add ioctl to call the function */
    // 执行加载到内核空间的函数，并将执行结果返回
    int result = ioctl(fd, JIT_IOCTL_EXECUTE_FUNC, 0);

    if (result == 42) {
        printf("Result is 42, success!\n");
    } else {
        printf("Failed, result is %d\n", result);
    }

    close(fd);
    return 0;
}
```

(2) 编译 (工作目录下)

在 `kernelJIT.c` 文件中的 TODO 部分完成代码编写后，
前往 `linux/tools/labs/skels/Kbuild` 目录下，添加 `kernelJIT` 目录，具体格式参照文件中的示例.
然后在工作目录 `linux/tools/labs` 下执行 `make build` 编译内核模块.
最后在 `linux/tools/labs/skels/lab4/kernelJIT` 目录下执行如下命令编译测试程序:

```
gcc -m32 --static -fno-pie -o testJIT ./testJIT.c
```

(如果遇到 `asm/ioctls.h` 找不到的错误，可以运行 `sudo ln -s /usr/include/asm-generic /usr/include/asm`)

(3) 测试程序运行 (虚拟环境下)

模块加载与设备节点创建 (虚拟环境下)
启动虚拟环境，加载内核模块.
通过 `cat /proc/devices` 查找字符设备的主设备号 (寻找名为 `JIT` 的设备)
执行以下命令创建设备节点:

```
mknod /dev/JIT c <主设备号> 0
```

测试程序运行 (虚拟环境下)
在虚拟环境中运行 `testJIT` 测试程序.
理想输出应为 `Result is 42, success!`

运行结果:

```
qemux86 login: root
root@qemux86:~# cd skels/lab4/kernelJIT
root@qemux86:~/skels/lab4/kernelJIT# insmod kernelJIT.ko
kernelJIT: loading out-of-tree module taints kernel.
kallsyms_lookup_name is d5f51aea
JIT: allocated executable page at c48a3000
JIT: module loaded
root@qemux86:~/skels/lab4/kernelJIT# lsmod | grep kernelJIT
kernelJIT 16384 0 - Live 0xe0876000 (O)
root@qemux86:~/skels/lab4/kernelJIT# cat /proc/devices
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 /dev/vc/0
  4 tty
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
 10 misc
 13 input
128 ptm
136 pts
229 hvc
252 JIT
253 virtio-portsdev
254 bsg

Block devices:
  7 loop
254 virtblk

Block devices:
  7 loop
254 virtblk
259 blkext
root@qemux86:~/skels/lab4/kernelJIT# mknod /dev/JIT c 252 0
root@qemux86:~/skels/lab4/kernelJIT# ls -l /dev/JIT
crw-r--r--    1 root     root      252,   0 Nov 10 04:18 /dev/JIT
root@qemux86:~/skels/lab4/kernelJIT# ./testJIT
Result is 42, success!
root@qemux86:~/skels/lab4/kernelJIT#
```

(4) 思考题

- 为什么在设置页面权限时先设置为只读 (RO) 再设置为可执行 (X),
  或者在需要写操作时先设置为不可执行 (NX) 再设置为可写 (RW)?

  答: 在操作系统中，为了保障安全性，执行权限和写权限不能同时存在于同一个内存页上.
  将内存页的权限从只读 (RO) 设置为可执行 (X) 或者从不可执行 (NX) 设置为可写 (RW),
  有助于防止恶意代码利用缓冲区溢出等漏洞进行攻击.
  例如攻击者如果能够执行可执行页的内容并写入数据，这可能导致远程执行代码.
  因此内存页权限的切换限制了执行与写入操作的同时进行，提高了系统的安全性.

- 为什么高版本内核中限制了 `set_memory_*` 函数在内核模块中的使用?

答: 在高版本内核中，限制 `set_memory_*` 函数的使用主要是出于安全性和稳定性的考虑.
`set_memory_*` 函数允许直接修改内存页面的权限，可能会被恶意内核模块滥用，
导致可执行内存区域被错误设置，增加系统的漏洞暴露面.
此外，动态修改内存页面的权限可能会导致内核崩溃或引发未定义的行为.
因此限制内核模块使用这些函数有助于提高内核的安全性和稳定性.

Task 3

3.1 背景

在本任务中，我们将使用 `userfaultfd` 实现一个简化版的数据仓库客户端.
数据仓库是一种专门用于存储和管理大量数据的系统，通常能够支持高效的数据读写，以便快速响应分析需求.
然而数据仓库中的数据量往往非常庞大，在数据分析时将整个数据仓库加载到本地并不现实，
因此按需加载成为了数据仓库客户端的普遍做法.

同样地，在操作系统内核中，按需加载 (或延迟加载) 机制也有广泛应用.
例如在进程创建时，系统不会预先分配大量堆空间，而是通过 `brk` 和 `mmap` 等系统调用在进程生命周期内按需扩展堆空间.
按需加载通常涉及 "页错误" (page fault):
当用户访问尚未加载的内存区域时，由于该区域还未映射到页表中，会触发页错误.
内核在处理页错误时会识别用户对该内存的需求，将该部分内存映射到页表中，从而完成按需加载的过程.

传统的页错误处理是在内核态进行的，对用户进程是透明的.
然而在用户态中也有按需加载的需求，用户有时希望自定义页错误的处理逻辑，
例如在本实验中，页错误处理逻辑是向服务器请求所需的数据.
Linux 提供的 `userfaultfd` 机制正是为支持用户态处理页错误而设计的，它体现了 "文件即一切" 的哲学.

`userfaultfd` 的核心是一个文件描述符，它由系统调用 `syscall(SYS_userfaultfd, O_CLOEXEC | O_NONBLOCK);` 返回.
通过这个文件描述符，我们可以通过文件操作的接口来控制内核的某些功能，例如:

- **① 设置监控区域:**
  通过 `mmap` 映射一块内存区域，然后使用 `ioctl(userfault_fd, UFFDIO_REGISTER, &uffd_reg)` 注册这块区域，
  通知内核监控该区域的页错误.
  `UFFDIO_REGISTER` 是一个 `ioctl` 命令，用于将指定的内存区域与 `userfaultfd` 关联，
  以便在该区域发生页错误时触发用户态处理.
- **② 处理页错误:**
  当注册的区域发生页错误时，我们可以通过 `poll` 等监视机制监听 `userfaultfd` 文件描述符.
  一旦检测到事件，通过 `read` 操作读取页错误的详细信息 (例如触发页错误的地址)，
  这些信息被封装在 `uffd_msg` 结构中，包括页错误的位置等关键数据，供用户态处理逻辑使用.
- **③ 提供缺页数据:**
  处理页错误后，用户态可以通过 `UFFDIO_COPY` 命令将所需数据传递给内核，以填充缺失的页.
  例如使用 `ioctl(userfault_fd, UFFDIO_COPY, &uffdio_copy)` 将用户态的数据缓冲区复制到内核空间，从而完成缺页的填充.

上述内容中提到了一些文件操作 (如 `poll` 和 `ioctl`)
`poll` 可理解为监听文件的操作，等待特定事件 (例如数据可读取，这里指有待处理的页错误) 发生.
`ioctl` 则是用于配置相关参数的操作，例如设置 `userfaultfd` 的监听内存区域.

本任务利用 `userfaultfd` 机制，实现了一个简易的数据仓库客户端，以按需读取数据仓库的数据.
代码的总体逻辑如下:

- 构建数据仓库的匿名映射:
  仅在虚拟内存中划定一片区域给数据仓库使用，这片区域不对应任何物理页，访问虚拟页时会触发页错误.
- 根据学号生成 5 个访问请求，访问数据仓库中的 5 个 "虚拟页".
  由于最初没有为数据仓库映射任何物理页，这 5 个访问请求必然会产生页错误.
- 这些页错误将由自定义的页错误处理函数处理，函数会向服务器请求对应的物理页并映射到虚拟内存中，从而完成按需加载.
  正确处理页错误后，即可访问对应的虚拟页并打印出虚拟页内容 (莎士比亚的文章)

3.2 实验任务

实现:

- ① `handle_pagefault` **函数:**
  考虑结构体 `uffd_msg`:

```
struct uffd_msg {
    __u8 event;  // 事件类型，值为 UFFD_EVENT_PAGEFAULT 表示发生页面错误
    union {
        struct {
            __u64 address;      // 缺页地址
            __u32 flags;        // 缺页的相关标志
            __u32 pad;
        } pagefault;
        ...
    } arg;
};
```

  - **确认页错误地址:**

```
uint64_t fault_addr = (uint64_t)uffd_msg->arg.pagefault.address;
```

    在页面错误事件发生时，`uffd_msg->arg.pagefault.address` 表示引发缺页错误的确切内存地址.
    也就是说，程序试图访问的这一地址并没有实际映射到物理内存.

  - **计算页号:**

```
uint32_t page_no = (fault_addr - file_base) / page_size;
```

    其中 `fault_addr` 是页面错误地址，而 `file_base` 是页面映射区域的基地址.

```
static void handle_pagefault(int uffd, struct uffd_msg *uffd_msg)
{
    // 首先检查 uffd_msg 消息事件是否为 UFFD_EVENT_PAGEFAULT
    // 如果不是该事件，程序会输出错误并退出
    if (uffd_msg->event != UFFD_EVENT_PAGEFAULT)
    {
        fprintf(stderr, "Unknown event on userfaultfd.\n");
        exit(EXIT_FAILURE);
```

```
    }

    // 打印发生页错误的地址
    printf("\e[0;32mPage fault detected at %lx\e[0m\n",
           (uint64_t)uffd_msg->arg.pagefault.address);

    /**
     * @todo Fetch the missing page from the server using
`fetch_remote_page`.
     * Determine the two arguments required:
     * - The destination buffer for the page content
     * * - The page number, calculated based on the faulting address and
page size
     */
    uint8_t *page = malloc(page_size);
    if (page == NULL) {
        fprintf(stderr, "Failed to allocate memory for page data.\n");
        exit(EXIT_FAILURE);
    }

    /* Calculate page number based on the fault address */
    uint64_t fault_addr = (uint64_t)uffd_msg->arg.pagefault.address;
    uint32_t page_no = (fault_addr - file_base) / page_size;

    /* Fetch the page from the server */
    fetch_remote_page(page, page_no);
    /* TODO END */

    struct uffdio_copy uffdio_copy;
    memset(&uffdio_copy, 0, sizeof(uffdio_copy));
    uffdio_copy.src = (uint64_t)page;
    uffdio_copy.dst = file_base + page_no * 4096;
    uffdio_copy.len = page_size;
    ioctl(uffd, UFFDIO_COPY, &uffdio_copy);
    if (uffdio_copy.copy != page_size)
    {
        fprintf(stderr, "Data only filled %" PRId64 "bytes \n",
                (long int)uffdio_copy.copy);
        exit(EXIT_FAILURE);
    }
    free(page);
}
```

- **② init_userfaultfd 函数:**
  考虑结构体 `uffdio_register`:

```
struct uffdio_register {
    struct uffd_range range;    /* 要监控的内存区域 */
    __aligned_u64 mode;         /* 监控模式 */
};
```

  - `range` 字段是一个类型为 `uffd_range` 的结构体.
    它定义了要监控的内存区域的起始地址和长度:

```
struct uffd_range {
    __aligned_u64 start;  /* 起始地址 */
    __aligned_u64 len;    /* 区域长度 */
};
```

`start`: 表示监控的内存区域的起始地址，而 `len` 表示监控区域的长度.

- `mode` 字段是一个 `__aligned_u64` 类型的值，用于指定监控该内存区域的模式.
  常见的模式有:
  - `UFFDIO_REGISTER_MODE_MISSING`: 表示仅当访问该区域的页面没有映射 (即发生缺页中断) 时，才会触发事件
  - `UFFDIO_REGISTER_MODE_WP`: 表示当该区域的页面变为写保护 (例如被标记为只读) 时，会触发事件
  - `UFFDIO_REGISTER_MODE_ACCESS`: 表示监控访问该区域的所有页面错误 (无论是读还是写)

配置 `uffdio_register` 的三个字段 `start`、`len` 和 `mode`:

- `uffdio_register.range.start` 指定了要注册的内存区域的起始地址:

```
uffdio_register.range.start = (uint64_t)addr;
```

- `uffdio_register.range.len` 指定了要监控的内存区域的长度:

```
uffdio_register.range.len = len;
```

- `uffdio_register.mode` 设置 `userfaultfd` 监控区域的模式:

```
uffdio_register.mode = UFFDIO_REGISTER_MODE_MISSING;
```

```
static int init_userfaultfd(uint8_t *addr, int len)
{
    int uffd = syscall(SYS_userfaultfd, O_CLOEXEC | O_NONBLOCK);

    /* Initialize userfaultfd, check for requested features. */
    struct uffdio_api uffdio_api;
    memset(&uffdio_api, 0, sizeof(uffdio_api));
    uffdio_api.api = UFFD_API;
    ioctl(uffd, UFFDIO_API, &uffdio_api);

    /* Register userfaultfd handler for addr region. */
    struct uffdio_register uffdio_register;
    memset(&uffdio_register, 0, sizeof(uffdio_register));

    /**
     * @todo Configure the three fields of `uffdio_register`: `start`,
`len`, and `mode`.
     * For detailed information on these fields and their usage, refer to
the documentation at:
     * https://man7.org/linux/man-pages/man2/UFFDIO_REGISTER.2const.html
     */
    uffdio_register.range.start = (uint64_t)addr;
    uffdio_register.range.len = len;
    uffdio_register.mode = UFFDIO_REGISTER_MODE_MISSING;
```

```
    /* TODO END */

    if (ioctl(uffd, UFFDIO_REGISTER, &uffdio_register) == -1) {
        perror("ioctl UFFDIO_REGISTER failed");
        exit(EXIT_FAILURE);
    }

    return uffd;
}
```

- ③ `main` **函数:**
  结构体 `pollfd` 用于指定要监控的文件描述符以及相关的事件类型:

```
struct pollfd {
    int fd;         /* 文件描述符 */
    short events;   /* 关注的事件 */
    short revents; /* 实际发生的事件 */
};
```

  - `fd` 是一个整数，表示要监控的文件描述符.
  - `events` 字段指定了我们希望监控的事件类型.
  - `revents` 字段由 `poll` 函数填充，它表示实际上发生的事件.

  配置 `pollfd` 的 `fd` 和 `events` 字段:

  - ```
    pollfd.fd = uffd;
    ```

    其中 `userfaultfd` 是一个用于监控用户空间内存缺页错误的特殊文件描述符
  - ```
    pollfd.events = POLLIN;
    ```

    `POLLIN` 事件指示文件描述符已经准备好进行读取操作，通常表示文件描述符上有数据可供读取.
    在 `userfaultfd` 的情况下， `POLLIN` 事件表示内存区域发生了页面错误，导致 `userfaultfd` 变为可读.
  - `revents` 字段由 `poll` 函数填充.
    调用 `poll` 后，可以检查这个字段来确定哪些事件已经发生:

    ```
    poll(&pollfd, 1, -1);
    ```

```
/* Handle page faults. */
for (;;)
{
    struct pollfd pollfd;
    memset(&pollfd, 0, sizeof(pollfd));
    /**
    * @todo Initialize `fd` and `events` fields of the `pollfd` structure.
    * We use `pollfd` and the `poll` function to monitor `userfaultfd`,
    * which becomes readable when a page fault event occurs.
    */
    pollfd.fd = uffd;
    pollfd.events = POLLIN;
    /* TODO END */
```

```
    poll(&pollfd, 1, -1);

    /**
    * After `poll` returns, `userfaultfd` is readable, signaling a page
fault event.
    * Read `uffd_msg` from `userfaultfd` to gather information for page
fault handling.
    */
    struct uffd_msg uffd_msg;
    if (read(uffd, &uffd_msg, sizeof(uffd_msg)) == 0)
    {
        fprintf(stderr, "Failed to read from uffd\n");
        exit(1);
    }

    handle_pagefault(uffd, &uffd_msg);
}
```

测试结果:

```
Linux:~/Desktop/OS/Lab2/linux/tools/labs/skels/lab4/Task3-Stu$ make
make: 'client' is up to date.
Linux:~/Desktop/OS/Lab2/linux/tools/labs/skels/lab4/Task3-Stu$ sudo ./client
[sudo] password for ycy:
Page fault detected at 7366173fc000
ive me another horse. Bind up my wounds.
    Have mercy, Jesu! Soft! I did but dream.
    O coward c
Page fault detected at 736617347000
IALLY.  PROHIBITED COMMERCIAL DISTRIBUTION INCLUDES BY ANY
SERVICE THAT CHARGES FOR DOWNLOAD TIME OR
Page fault detected at 7366173ce000
rs; but my time
    Runs posting on in Bolingbroke's proud joy,
    While I stand fooling here, his
Page fault detected at 736617406000
heir breaths with sweetmeats tainted are.
    Sometime she gallops o'er a courtier's nose,
    And t
Page fault detected at 736617102000
h me. I will go seek the King.
    This is the very ecstasy of love,
    Whose violent property ford
Linux:~/Desktop/OS/Lab2/linux/tools/labs/skels/lab4/Task3-Stu$ nc -w 5 10.20.26.33 38324
21307140051
Page fault detected at 7f199f4fa000
ive me another horse. Bind up my wounds.
    Have mercy, Jesu! Soft! I did but dream.
    O coward c
Page fault detected at 7f199f445000
IALLY.  PROHIBITED COMMERCIAL DISTRIBUTION INCLUDES BY ANY
SERVICE THAT CHARGES FOR DOWNLOAD TIME OR
Page fault detected at 7f199f4cc000
rs; but my time
    Runs posting on in Bolingbroke's proud joy,
    While I stand fooling here, his
Page fault detected at 7f199f504000
heir breaths with sweetmeats tainted are.
    Sometime she gallops o'er a courtier's nose,
    And t
Page fault detected at 7f199f200000
h me. I will go seek the King.
    This is the very ecstasy of love,
    Whose violent property ford
Linux:~/Desktop/OS/Lab2/linux/tools/labs/skels/lab4/Task3-Stu$
```

The End