

华中科技大学

本科生课程设计报告

课 程 : 操作系统原理设计与实践

题 目 : 阻塞设备驱动开发与内核调试

院 系 : 软件学院

专业班级 : 软件工程 2102 班

学 号 : U2021172??

姓 名 : dekrt

2023 年 06 月 09 日

目 录

1	任务一 阻塞设备驱动开发	1
1.1	实验任务概述	1
1.2	实验设计思路	2
1.3	程序的重点/难点/核心技术分析	5
1.4	运行和测试过程	7
1.5	实验心得和建议	11
1.6	学习和编程实现参考网址	12
2	任务二 理解、跟踪和调试页式虚拟内存管理机制	13
2.1	实验任务概述	13
2.2	页式虚拟内存管理机制流程描述	13
2.3	内核跟踪和调试程序设计	18
2.4	运行和测试过程	19
2.5	实验心得和建议	21
2.6	学习和编程实现参考网址	22

1 任务一 阻塞设备驱动开发

1.1 实验任务概述

本课程设计实验的主要任务是编写驱动程序，支持多个测试程序对内核缓冲区的有序读/写。实验的目的是深化对操作系统原理的理解，提高内核编程技巧，以及提升解决实际问题的能力。

具体的实验工作和开发步骤如下：

1. 定义内核缓冲区 KFIFO（大小可调，例如设定为 32 字节）和等待队列。
该缓冲区是环形缓冲区，由驱动程序维护两个读写指针。
2. 缓冲区按序读写，确保每个数据的读/写不重复，不遗漏。
3. 编写若干测试程序读或写缓冲区的若干字节。具体要求包括：
 - 读操作时：当缓冲区有足够的数据可读时，执行读操作；否则，读进程被阻塞，直到有足够的数据可读时才被唤醒。
 - 写操作时：当缓冲区有足够的空位可写时，执行写操作；否则，写进程被阻塞，直到有足够的空位可写时才被唤醒。
4. 驱动程序负责维护缓冲区的读/写，并根据情况适时阻塞或唤醒相应的进程。
5. 在实验过程中，需要观察和分析缓冲区变化、测试程序的阻塞/唤醒情况、阻塞队列等相关情况。

通过完成这些任务，我们可以更深入地理解操作系统的工作原理，尤其是进程间通信、同步与互斥等关键概念。

1.2 实验设计思路

1 整体结构

本实验的设计思路主要分为三个部分：设备驱动程序的编写，测试程序的编写，以及这两部分之间的交互。

设备驱动程序（`my_dev.c`）的主要任务是定义和管理内核 FIFO 缓冲区，以及处理与之相关的读写操作。这部分代码还定义了一个互斥锁（Mutex）用于同步对 FIFO 缓冲区的访问，以及两个等待队列（`read_queue` 和 `write_queue`）用于在 FIFO 缓冲区满或空时阻塞读写进程。

测试程序包括 `produce.c` 和 `consume.c` 两部分。`produce.c` 负责向设备写入数据，`consume.c` 负责从设备读取数据。这两个程序通过系统调用 `open`、`read`、`write` 和 `close` 与设备驱动程序交互。

整个系统的整体结构图如图 1 所示：

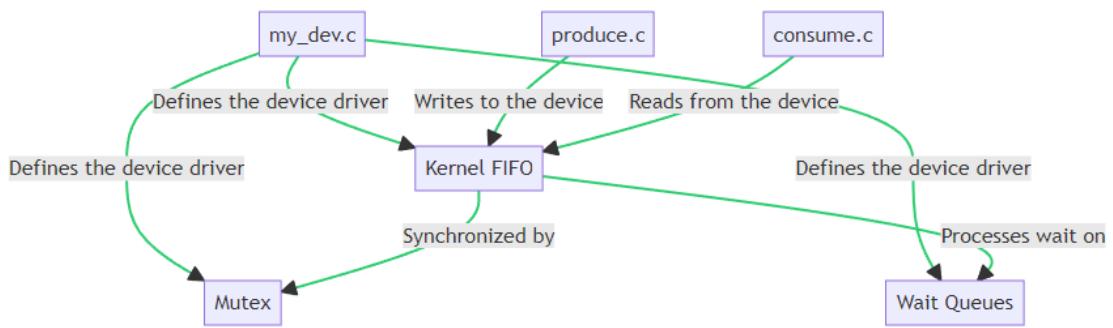


图 1：实验一系统的整体结构图

如图所示，设备驱动程序（`my_dev.c`）负责管理内核 FIFO 缓冲区、互斥锁和等待队列。测试程序（`produce.c` 和 `consume.c`）通过系统调用与设备驱动程序交互，实现对 FIFO 缓冲区的读写操作。

同时，为了测试程序时更加方便，我们也特地编写了 `Makefile` 文件，在使

用 make 命令后可以一并编译生产者和消费者的测试程序，方便测试，如图二所示：

```
Makefile
1 obj-m += my_dev.o
2 CC = gcc
3 PRODUCE_SRC = $(wildcard produce*.c)
4 PRODUCE_BIN = $(patsubst %.c, %, $(PRODUCE_SRC))
5 CONSUME_SRC = $(wildcard consume*.c)
6 CONSUME_BIN = $(patsubst %.c, %, $(CONSUME_SRC))
7
8 all: $(PRODUCE_BIN) $(CONSUME_BIN)
9     make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) modules
10
11 $(PRODUCE_BIN): %: %.c
12     $(CC) $(CFLAGS) -o $@ $<
13
14 $(CONSUME_BIN): %: %.c
15     $(CC) $(CFLAGS) -o $@ $<
16
17 clean:
18     make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) clean
19     rm -f $(PRODUCE_BIN)
20     rm -f $(CONSUME_BIN)
21
22 CONFIG_MODULE_SIG=n
```

图 2：Makfile 源代码

2 依赖的硬件环境或库函数

- <linux/miscdevice.h>：用于注册和注销杂项设备。
- <linux/kfifo.h>：提供内核 FIFO 的操作函数。
- <linux/wait.h>和<linux/sched.h>：提供等待队列的操作函数，用于阻塞和唤醒进程。
- <linux/mutex.h>：提供互斥锁的操作函数，用于同步。
- <linux/module.h>：提供模块的基本操作函数。

3 主要数据结构：

- **FIFOBuffer**：一个内核 FIFO，用于存储设备的数据。
- **Mutex**：一个互斥锁，用于同步对 FIFO 的访问。
- **read_queue** 和 **write_queue**：两个等待队列，分别用于阻塞读进程和写进程。

- **DevFops**: 一个文件操作结构，定义了设备的打开、关闭、读和写操作。
- **miscDeviceFIFOBlock**: 一个杂项设备结构，定义了设备的名称、设备号和文件操作。

4 主要变量:

- **my_misicdevice**: 一个设备结构指针，指向注册的杂项设备。
- **buf**: 一个字符数组，用于从 FIFO 中读取数据。
- **copied**: 一个无符号整数，表示从 FIFO 中复制的字节数。
- **read** 和 **write**: 两个整数，分别表示从 FIFO 中读取的字节数和写入 FIFO 的字节数。
- **ret**: 一个整数，表示函数的返回值。

5 程序的流程:

- 在模块初始化函数 **DevInit** 中，注册杂项设备，初始化互斥锁和等待队列。
- 在设备打开函数 **my_open** 中，打印设备打开的信息，并打印 FIFO 的内容。
- 在设备关闭函数 **my_release** 中，打印设备关闭的信息。
- 在设备读函数 **my_read** 中，如果 FIFO 为空，则阻塞读进程，否则从 FIFO 中读取数据，然后唤醒任何被阻塞的写进程。
- 在设备写函数 **my_write** 中，如果 FIFO 已满，则阻塞写进程，否则向 FIFO 中写入数据，然后唤醒任何被阻塞的读进程。
- 在模块退出函数 **DevExit** 中，注销杂项设备，并打印设备退出的信息。

6 注意事项

在实现过程中，我们需要注意以下几点：

1. 在读写操作中，我们需要检查 FIFO 缓冲区的状态。如果缓冲区满或空，

我们需要阻塞相应的读写进程，并将其加入到等待队列中。

2. 在 FIFO 缓冲区状态改变时（例如，从满变为非满，或从空变为非空），我们需要唤醒等待队列中的进程。
3. 在进行读写操作时，我们需要使用互斥锁来同步对 FIFO 缓冲区的访问，防止数据竞争。

以上就是本实验的设计思路。通过实现这个系统，我们可以更深入地理解操作系统中的进程管理和资源调度。

1.3 程序的重点/难点/核心技术分析

在设计和实现过程中，我们遇到了一些技术重点和难点，以下是对这些问题的总结和解决方法：

1 FIFO 缓冲区的管理

在 Linux 内核中，我们使用 kfifo API 来定义和管理 FIFO 缓冲区。这是一个环形缓冲区，我们需要维护两个读写指针。为了确保对 FIFO 缓冲区的安全访问，我们使用了互斥锁进行同步。如图 3 所示。

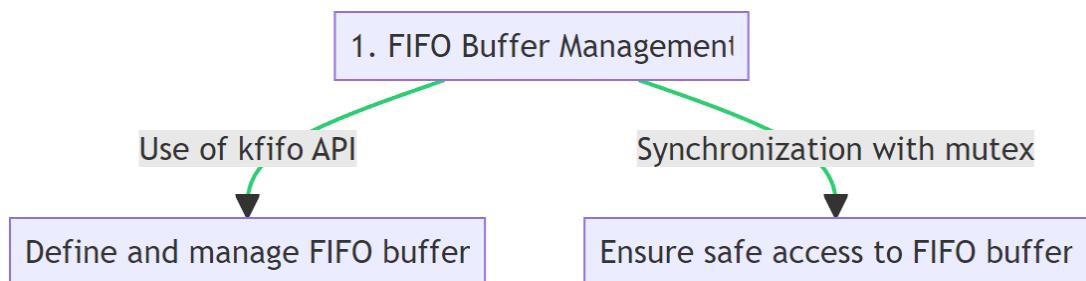


图 3： FIFO 缓冲区的管理

2 进程的阻塞和唤醒

当 FIFO 缓冲区满或空时，我们需要阻塞相应的读写进程，并将其加入到等待队列中。当 FIFO 缓冲区状态改变时（例如，从满变为非满，或从空变为非

空), 我们需要唤醒等待队列中的进程。这是一个重要的资源调度问题, 如图 4 所示。

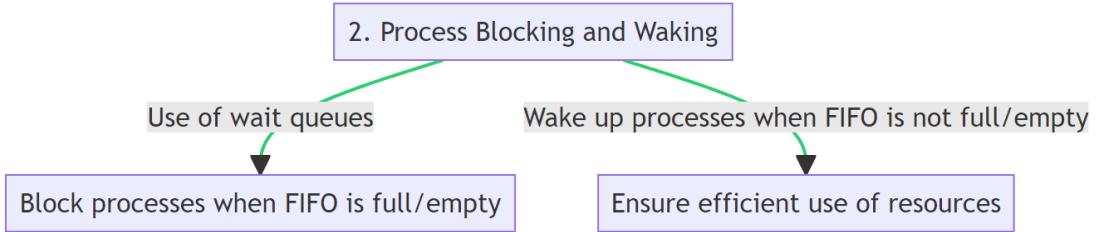


图 4: 进程的阻塞和唤醒

3 驱动程序和测试程序之间的交互

测试程序 (produce.c 和 consume.c) 通过系统调用与设备驱动程序交互, 实现对 FIFO 缓冲区的读写操作。我们需要处理阻塞和非阻塞操作, 以确保测试程序的正确行为。如图 5 所示。

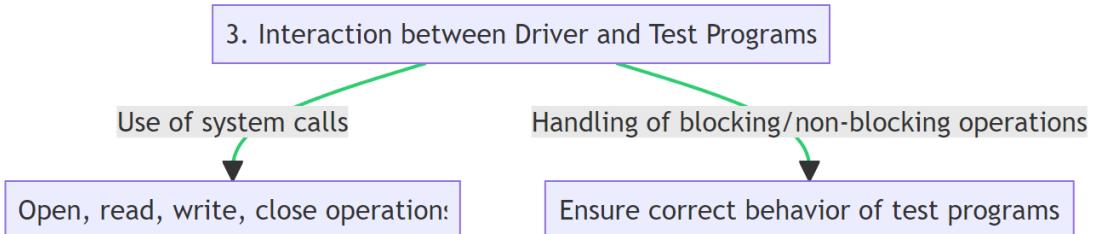


图 5: 驱动程序和测试程序之间的交互

4 错误处理

在编写代码时, 我们需要检查函数的返回值, 处理可能的错误。我们使用 printk 函数进行调试, 以识别和修复问题。如图 6 所示。

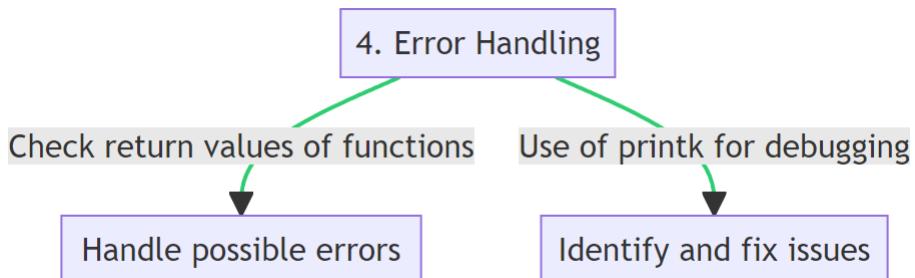


图 6: 错误处理

5 并发控制

在多进程环境中，我们需要使用互斥锁来防止数据竞争。我们还使用等待队列来阻塞和唤醒进程，以实现有效的资源调度。如图 7 所示。

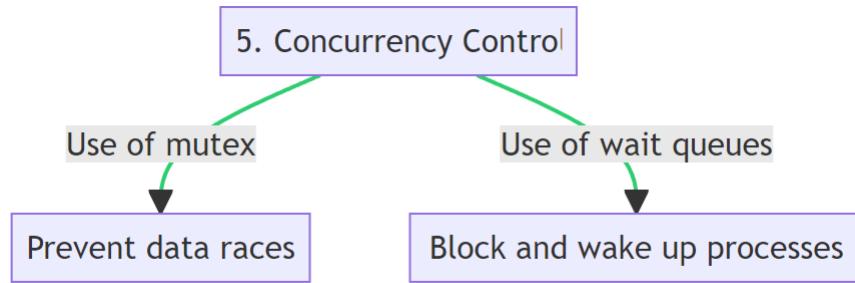


图 7：并发控制

以上就是本实验的重点和难点，以及我们的解决方法。通过解决这些问题，我们可以更深入地理解操作系统中的进程管理和资源调度。

1.4 运行和测试过程

1 编译和加载驱动程序

首先，需要编译和加载设备驱动程序。这可以通过使用 `make` 命令来完成。然后，使用 `sudo insmod my_dev.ko` 命令来加载驱动程序。然后使用 `lsmod` 命令查看驱动程序是否加载成功，如图 8 所示。

```
dekrt@dekrts-virtual-machine:~/code/Task1/Task1$ make
gcc -o produce1 produce1.c
gcc -o produce2 produce2.c
gcc -o produce3 produce3.c
gcc -o consume3 consume3.c
gcc -o consume2 consume2.c
gcc -o consume1 consume1.c
make -C /lib/modules/5.15.111/build M=/home/dekrts/code/Task1/Task1 modules
make[1]: 进入目录“/usr/src/linux-5.15.111”
  CC [M]  /home/dekrts/code/Task1/Task1/my_dev.o
  MODPOST /home/dekrts/code/Task1/Task1/Module.symvers
  CC [M]  /home/dekrts/code/Task1/Task1/my_dev.mod.o
  LD [M]  /home/dekrts/code/Task1/Task1/my_dev.ko
  BTF [M] /home/dekrts/code/Task1/Task1/my_dev.ko
make[1]: 离开目录“/usr/src/linux-5.15.111”
dekrt@dekrts-virtual-machine:~/code/Task1/Task1$ sudo insmod my_dev.ko
[sudo] dekrts 的密码:
dekrt@dekrts-virtual-machine:~/code/Task1/Task1$ lsmod
Module           Size  Used by
my_dev          16384  0
isofs            49152  1
rfcomm           81920  4
bnep             28672  2
nls_iso8859_1   16384  1
```

图 8：加载设备驱动程序

使用 `dmesg` 命令可以查看内核调试信息，会提示设备初始化成功，如图 9 所示。

```
[ 3.896273] audit: type=1400 audit(1684659671.668:9): apparmor="STATUS" operation="profile_load" profile="unconfined" name="nvidia_modprobe//kmod" pid=757 comm="apparmor_parser"
[ 3.897719] audit: type=1400 audit(1684659671.668:10): apparmor="STATUS" operation="profile_load" profile="unconfined" name="/usr/lib/NetworkManager/nm-dhcp-client.action" pid=756 comm="apparmor_parser"
[ 4.177239] Bluetooth: BNEP (Ethernet Emulation) ver 1.3
[ 4.177243] Bluetooth: BNEP filters: protocol multicast
[ 4.177248] Bluetooth: BNEP socket layer initialized
[ 4.291297] e1000: ens33 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: None
[ 4.292218] IPv6: ADDRCONF(NETDEV_CHANGE): ens33: link becomes ready
[ 4.983563] Bluetooth: RFCOMM TTY layer initialized
[ 4.983571] Bluetooth: RFCOMM socket layer initialized
[ 4.983577] Bluetooth: RFCOMM ver 1.11
[ 5.306342] loop11: detected capacity change from 0 to 8
[ 8.642719] ISO 9660 Extensions: Microsoft Joliet Level 3
[ 8.642947] ISO 9660 Extensions: RRIP_1991A
[ 8.669527] rfkill: input handler disabled
[ 60.693707] my_dev: loading out-of-tree module taints kernel.
[ 60.693743] my_dev: module verification failed: signature and/or required key missing - tainting kernel
[ 60.694478] my_dev: Init successfully
root@dekrts-virtual-machine:/home/dekrts/code/Task1/Task1#
```

图 9：内核提示信息

2 运行测试程序

接下来，在一个终端中运行 `produce1.c`，在另一个终端中运行 `consume1.c`。

这将模拟多个进程同时读写 FIFO 缓冲区的情况，如图 10 所示。

```
活动 终端 Player(P) | || - 活动 dekrt@dekrt-virtual-machine:~/code/Task1$ sudo su
[sudo] dekrt 的密码:
root@dekrt-virtual-machine:/home/dekrt... sudo ./.produce1
produce1 pid = 2570 申请写入12字符, 实际写入12字符
produce1 pid = 2570 申请写入8字符, 实际写入8字符
produce1 pid = 2570 申请写入12字符, 实际写入12字符
produce1 pid = 2570 申请写入12字符, 实际写入12字符
produce1 pid = 2570 申请写入8字符, 实际写入8字符
produce1 pid = 2570 申请写入7字符, 实际写入7字符
produce1 pid = 2570 申请写入6字符, 实际写入6字符
produce1 pid = 2570 申请写入11字符, 实际写入10字符
produce1 pid = 2570 申请写入9字符, 实际写入9字符
produce1 pid = 2570 申请写入8字符, 实际写入8字符
produce1 pid = 2570 申请写入7字符, 实际写入7字符
produce1 pid = 2570 申请写入6字符, 实际写入6字符
produce1 pid = 2570 申请写入10字符, 实际写入10字符
produce1 pid = 2570 申请写入8字符, 实际写入8字符
produce1 pid = 2570 申请写入7字符, 实际写入7字符
produce1 pid = 2570 申请写入6字符, 实际写入6字符
produce1 pid = 2570 申请写入5字符, 实际写入5字符
produce1 pid = 2570 申请写入8字符, 实际写入8字符
produce1 pid = 2570 申请写入7字符, 实际写入7字符
produce1 pid = 2570 申请写入9字符, 实际写入9字符

dekrt@dekrt-virtual-machine:~/code/Task1$ ./consume1
[sudo] dekrt 的密码:
open /dev/my_dev failed
dekrt@dekrt-virtual-machine:~/code/Task1$ sudo ./consume1
consume1 pid = 2581 申请读出12字符, 实际读出12字符
consume1 pid = 2581 申请读出11字符, 实际读出11字符
consume1 pid = 2581 申请读出6字符, 实际读出6字符
consume1 pid = 2581 申请读出8字符, 实际读出8字符
consume1 pid = 2581 申请读出6字符, 实际读出6字符
consume1 pid = 2581 申请读出12字符, 实际读出12字符
consume1 pid = 2581 申请读出7字符, 实际读出7字符
consume1 pid = 2581 申请读出9字符, 实际读出9字符
consume1 pid = 2581 申请读出6字符, 实际读出6字符
consume1 pid = 2581 申请读出10字符, 实际读出10字符
consume1 pid = 2581 申请读出7字符, 实际读出7字符
consume1 pid = 2581 申请读出8字符, 实际读出8字符
consume1 pid = 2581 申请读出7字符, 实际读出7字符
consume1 pid = 2581 申请读出8字符, 实际读出8字符
consume1 pid = 2581 申请读出8字符, 实际读出8字符
```

图 10：同时运行生产者、消费者程序

3 观察输出

使用 `dmesg -w` 命令查看 **kernel message**，驱动程序应该会打印任何阻塞或唤醒的进程。测试程序应该会打印出它们尝试读写的字节数，以及实际读写的字节数，如图 11 所示。

```

活动 终端 Player(P) zh Ubuntu 64位 zh
root@dekr-virtual-machine: /home/dekr...
produce1 pid = 2570 申请写入10字符, 实际写入6字符
produce1 pid = 2570 申请写入9字符, 实际写入9字符
produce1 pid = 2570
produce1 pid = 2570
produce1 pid = 2570 [ 189.955004] my_dev: Wake up read process, cmd = produce2
produce1 pid = 2570 [ 190.799901] my_dev: Wake up write process, cmd = consume1
produce1 pid = 2570 [ 190.799907] my_dev: Wake up read process, cmd = produce1
produce1 pid = 2570 [ 190.955933] my_dev: Wake up read process, cmd = produce2
produce1 pid = 2570 [ 191.800936] my_dev: Write process is blocked, needs space: 9, available space: 0
produce1 pid = 2570 [ 191.800939] my_dev: Wake up write process, cmd = consume1
produce1 pid = 2570 [ 191.800945] my_dev: Wake up read process, cmd = produce1
produce1 pid = 2570 [ 191.956893] my_dev: Write process is blocked, needs space: 8, available space: 0
produce1 pid = 2570 [ 192.801916] my_dev: Write process is blocked, needs space: 7, available space: 0
produce1 pid = 2570 [ 192.801965] my_dev: Wake up read process, cmd = produce1
produce1 pid = 2570 [ 192.802007] my_dev: Wake up read process, cmd = produce2
produce1 pid = 2570 [ 193.802861] my_dev: Write process is blocked, needs space: 7, available space: 0
produce1 pid = 2570 [ 193.802868] my_dev: Write process is blocked, needs space: 7, available space: 0
produce1 pid = 2570 [ 193.802874] my_dev: Wake up write process, cmd = consume1
produce1 pid = 2570 [ 193.802895] my_dev: Wake up read process, cmd = produce2
produce1 pid = 2570 [ 194.804005] my_dev: Write process is blocked, needs space: 6, available space: 0
produce1 pid = 2570 [ 194.804007] my_dev: Wake up write process, cmd = consume1
produce1 pid = 2570 [ 194.804012] my_dev: Wake up read process, cmd = produce2
produce1 pid = 2570 [ 195.804768] my_dev: Write process is blocked, needs space: 11, available space: 0
produce1 pid = 2570 [ 195.805261] my_dev: Wake up write process, cmd = consume1
produce1 pid = 2570 [ 195.805290] my_dev: Wake up read process, cmd = produce2
produce1 pid = 2570 [ 196.806195] my_dev: Write process is blocked, needs space: 9, available space: 0
produce1 pid = 2570 [ 196.806197] my_dev: Wake up write process, cmd = consume1
produce1 pid = 2570 [ 196.806201] my_dev: Wake up read process, cmd = produce2
produce1 pid = 2570 [ 197.806891] my_dev: Wake up write process, cmd = consume1
produce1 pid = 2570 [ 197.806949] my_dev: Wake up read process, cmd = produce1
produce1 pid = 2570 [ 197.807358] my_dev: Wake up read process, cmd = produce2
produce1 pid = 2570
produce1 pid = 2570 申请写入7字符, 实际写入7字符
produce1 pid = 2570 申请写入8字符, 实际写入8字符
produce1 pid = 2570 申请写入9字符, 实际写入7字符
produce1 pid = 2570 申请写入7字符, 实际写入7字符
produce1 pid = 2570 申请写入7字符, 实际写入7字符
consume1 pid = 2581 申请读出6字符, 实际读出6字符
consume1 pid = 2581 申请读出6字符, 实际读出6字符
consume1 pid = 2581 申请读出10字符, 实际读出10字符
consume1 pid = 2581 申请读出10字符, 实际读出9字符
consume1 pid = 2581 申请读出9字符, 实际读出9字符
consume1 pid = 2581 申请读出12字符, 实际读出12字符

```

图 11：在内核信息中打印唤醒进程

4 卸载驱动程序

完成测试后，可以使用 `rmmmod` 命令来卸载驱动程序。

```

root@dekr-virtual-machine:/home/dekr/code/Task1/Task1# rmmmod my_dev.
rmmmod: ERROR: Module my_dev is in use
root@dekr-virtual-machine:/home/dekr/code/Task1/Task1# kill --help
kill: kill [-s 信号声明 | -n 信号编号 | -信号声明] 进程号 | 任务声明 ... 或 kill -l [信号
声明]
向一个任务发送一个信号。
向以 PID 进程号或者 JOBSPEC 任务声明指定的进程发送一个以
SIGSPEC 信号声明或 SIGNUM 信号编号命名的信号。如果没有指定
SIGSPEC 或 SIGNUM，那么假定发送 SIGTERM 信号。
选项：
-s sig    SIG 是信号名称
-n sig    SIG 是信号编号
-l        列出信号名称；如果参数后跟 `-' 则被假设为信号编号，
而相应的信号名称会被列出
Kill 成为 shell 内建有两个理由：它允许使用任务编号而不是进程号，
并且在可以创建的进程数上限达到时允许进程被杀死。
退出状态：
返回成功，除非使用了无效的选项或者有错误发生。
root@dekr-virtual-machine:/home/dekr/code/Task1/Task1# kill 2581
root@dekr-virtual-machine:/home/dekr/code/Task1/Task1# kill 2570
root@dekr-virtual-machine:/home/dekr/code/Task1/Task1# kill 2595
root@dekr-virtual-machine:/home/dekr/code/Task1/Task1# lsmod | grep my_
my_dev          16384  0
root@dekr-virtual-machine:/home/dekr/code/Task1/Task1# rmmmod my_dev.ko
root@dekr-virtual-machine:/home/dekr/code/Task1/Task1#

```

图 12：卸载设备驱动程序

1.5 实验心得和建议

在本实验中，我遇到了一些挑战和问题，以下是我对这些问题的总结和解决方法：

1 环境配置问题

在开始实验之前，我需要配置适当的 Linux 环境。我选择了 Ubuntu 作为我的操作系统，因为它对新手友好，有大量的在线资源可以参考。我遇到的问题是缺少一些必要的开发工具和库，例如 GCC 和 Linux 内核头文件。我通过使用 `sudo apt-get install` 命令来安装这些工具和库，并使用 `sudo apt-get update` 对软件包进行更新，解决了这个问题。

2 命令操作问题

在编译和加载驱动程序时，我遇到了一些问题。我最初忘记了使用 `sudo` 命令来获取必要的权限，程序提示：无法访问设备：`/dev/my_dev`。我通过查阅文档和在线资源，了解到我需要使用 `sudo` 命令来执行这些操作。使用 `sudo` 命令执行测试程序后，问题解决。

3 编译错误

在编写驱动程序时，我遇到了一些编译错误。如在执行 `make` 时，会报错：

```
make[1]: *** No rule to make target  
'arch/x86/entry/syscalls/syscall_32.tbl', needed by  
'arch/x86/entry/syscalls/../../include/generated/asm/syscalls_32.h'.  
Stop.  
make: *** [archheaders] Error 2
```

我通过仔细检查我的代码，在 `Stackoverflow` 上进行搜索，对我的 `Makefile` 文件做了如下的修改：将

```
$(MAKE) -C $(KDIR) M=$(PWD) modules
```

修改为

```
$(MAKE) -C $(KDIR) M=$(shell pwd) modules
```

随后再运行 `make` 指令，没有报错。

4 运行错误

在运行测试程序时，我遇到了一些运行错误。这些错误主要是由于我没有正确地处理阻塞和唤醒进程。我通过查阅文档和在线资源，了解到我需要使用等待队列来实现这个功能。

总的来说，这个实验是一个很好的学习经验。我不仅学习了如何编写 `Linux` 设备驱动程序，还学习了如何处理并发和资源调度问题。我建议未来的学生成在开始实验之前，先熟悉 `Linux` 环境和基本的命令操作。此外，他们应该仔细阅读文档和在线资源，以了解如何使用 `kfifo API` 和等待队列。

1.6 学习和编程实现参考网址

1. [Linux Kernel Documentation](#) 这是 Linux 内核的官方文档，包含了大量的信息和示例代码。我主要参考了关于 `kfifo API` 和等待队列的部分。
2. [Linux Device Drivers, Third Edition](#) 这是一本关于 Linux 设备驱动程序的经典书籍。我参考了其中的一些章节，以了解如何编写设备驱动程序。
3. [Stack Overflow](#) 这是一个程序员问答网站，我在这里找到了一些关于如何处理并发和资源调度问题的解决方案。
4. [Linux Journal: Tech Tip: Kernel Debugging with Kfifo](#) 这篇文章介绍了如何使用 `kfifo` 进行内核调试，我参考了其中的一些技巧和建议。
5. [GitHub](#) 我在 GitHub 上找到了一些相关的开源项目和代码示例，这些项目和代码示例帮助我理解如何在实际项目中使用 `kfifo API` 和等待队列。

2 任务二 理解、跟踪和调试页式虚拟内存管理机制

2.1 实验任务概述

本次课设实验的任务是理解、跟踪和调试页式虚拟内存管理机制。实验的目的在于通过阅读内核源代码，跟踪、调试和优化页式内存管理模块，深入理解操作系统中的内存管理机制。

具体工作包括阅读内核源代码，跟踪页式内存管理模块，理解哪些函数与页面/页框的管理有关，哪个函数负责分配页框，哪个函数处理缺页，哪个函数调整页表，并画出相关函数的执行流程图。

此外，还需要修改内核，检测特定应用程序（例如：**HustSSE**，可修改）调入内存后占用了哪些页框，并使用 **printk** 函数输出；同时，也需要检测页框何时被释放，并使用 **printk** 函数输出。只要该特定应用程序运行，内核就会输出相关信息。

通过这次实验，可以帮助我们更好地理解和掌握操作系统中的内存管理机制，提高我们的编程能力和解决实际问题的能力。

2.2 页式虚拟内存管理机制流程描述

实验中我阅读了 Linux 5.15.111 的内核源代码，以下是与页面/页框管理相关的一些函数：

首先，我发现了 **_alloc_pages_nodemask** 函数。这个函数的主要任务是分配页框。它首先尝试在给定的内存域（node）和内存区域（zone）中分配页框。如果在这些区域内无法成功分配，它会尝试使用备用节点和区域进行分配。这种设

计使得内存分配更加灵活，能够在不同的内存区域之间进行平衡，如图 13 所示。

```
5380  /*
5381   * This is the 'heart' of the zoned buddy allocator.
5382   */
5383 struct page *_alloc_pages(gfp_t gfp, unsigned int order, int preferred_nid,
5384                           nodemask_t *nodemask)
5385 {
5386     struct page *page;
5387     unsigned int alloc_flags = ALLOC_WMARK_LOW;
5388     gfp_t alloc_gfp; /* The gfp_t that was actually used for allocation */
5389     struct alloc_context ac = { };
5390
5391     /*
5392      * There are several places where we assume that the order value is sane
5393      * so bail out early if the request is out of bound.
5394      */
5395     if (unlikely(order >= MAX_ORDER)) {
5396         WARN_ON_ONCE(!(gfp & __GFP_NOWARN));
5397         return NULL;
5398     }
5399
5400     gfp &= gfp_allowed_mask;
5401     /*
5402      * Apply scoped allocation constraints. This is mainly about GFP_NOFS
5403      * resp. GFP_NOIO which has to be inherited for all allocation requests
```

图 13: `_alloc_pages` 函数的源代码示意图

接下来，我研究了 `handle_mm_fault` 函数。这个函数的职责是处理缺页异常。

当一个进程试图访问一个尚未映射到其地址空间的页，或者没有足够权限访问的页时，就会发生缺页异常。`handle_mm_fault` 函数会尝试解决这个问题，可能的解决方式包括分配新的页框，或者改变现有页的权限。这个函数的存在保证了系统的稳定性和安全性，如图 14 所示。

```

4654 * The mmap_lock may have been released depending on flags and our
4655 * return value. See filemap_fault() and __lock_page_or_retry().
4656 */
4657 static vm_fault_t __handle_mm_fault(struct vm_area_struct *vma,
4658         unsigned long address, unsigned int flags)
4659 {
4660     struct vm_fault vmf = {
4661         .vma = vma,
4662         .address = address & PAGE_MASK,
4663         .flags = flags,
4664         .pgoff = linear_page_index(vma, address),
4665         .gfp_mask = __get_fault_gfp_mask(vma),
4666     };
4667     unsigned int dirty = flags & FAULT_FLAG_WRITE;
4668     struct mm_struct *mm = vma->vm_mm;
4669     pgd_t *pgd;
4670     p4d_t *p4d;
4671     vm_fault_t ret;
4672
4673     pgd = pgd_offset(mm, address);
4674     p4d = p4d_alloc(mm, pgd, address);
4675     if (!p4d)
4676         return VM_FAULT_OOM;
4677
4678     vmf.pud = pud_alloc(mm, p4d, address);

```

图 14: __handle_mm_fault 函数的源代码示意图

我还发现了 `pte_alloc_one` 和 `pte_free` 两个函数，它们分别用于分配和释放页表项 (Page Table Entry)。页表项是一种重要的数据结构，它用于映射虚拟地址到物理地址。这两个函数是虚拟内存管理的基础，如图 15 所示。

```

48 * __pte_alloc_one - allocate a page for PTE-level user page table
49 * @mm: the mm_struct of the current context
50 * @gfp: GFP flags to use for the allocation
51 *
52 * Allocates a page and runs the pgtable_pte_page_ctor().
53 *
54 * This function is intended for architectures that need
55 * anything beyond simple page allocation or must have custom GFP flags.
56 *
57 * Return: `struct page` initialized as page table or %NULL on error
58 */
59 static inline pgtable_t __pte_alloc_one(struct mm_struct *mm, gfp_t gfp)
60 {
61     struct page *pte;
62
63     pte = alloc_page(gfp);
64     if (!pte)
65         return NULL;
66     if (!pgtable_pte_page_ctor(pte)) {
67         __free_page(pte);
68         return NULL;
69     }
70
71     return pte;
72 }

```

图 15: pte_alloc_one 函数的源代码示意图

此外，我还注意到了 `_SetPageReserved` 和 `_ClearPageReserved` 两个函数。这

两个函数用于设置和清除页框的保留标志。保留的页框不能被页框分配器分配。

这种设计可以保护关键的内存区域，防止它们被错误地分配或释放，如图 16 所示。

```
360 /* Xen */
361 PAGEFLAG(Pinned, pinned, PF_NO_COMPOUND)
362 | TESTSCFLAG(Pinned, pinned, PF_NO_COMPOUND)
363 PAGEFLAG(SavePinned, savepinned, PF_NO_COMPOUND);
364 PAGEFLAG(Foreign, foreign, PF_NO_COMPOUND);
365 PAGEFLAG(XenRemapped, xen_remapped, PF_NO_COMPOUND)
366 | TESTCLEARFLAG(XenRemapped, xen_remapped, PF_NO_COMPOUND)
367 |?
368 PAGEFLAG(Reserved, reserved, PF_NO_COMPOUND)
369 |? CLEARPAGEFLAG(Reserved, reserved, PF_NO_COMPOUND)
370 |? SETPAGEFLAG(Reserved, reserved, PF_NO_COMPOUND)
371 PAGEFLAG(SwapBacked, swapbacked, PF_NO_TAIL)
372 |? CLEARPAGEFLAG(SwapBacked, swapbacked, PF_NO_TAIL)
373 |? SETPAGEFLAG(SwapBacked, swapbacked, PF_NO_TAIL)
374 |
375 /*
376 * Private page markings that may be used by the filesystem that owns the
377 * for its own purposes.
378 * - PG_private and PG_private_2 cause releasepage() and co to be invoked
379 */
```

图 16: __SetPageReserved 函数的源代码示意图

最后，我研究了 `free_hot_cold_page` 和 `free_hot_cold_page_list` 两个函数。这两个函数用于释放页框，将其返回到页框分配器。这样可以确保系统的内存资源得到有效的利用，如图 17 所示。

```
void free_hot_cold_page(struct page *page, bool cold)
{
    struct zone *zone = page_zone(page); // 用于根据page得到所在zone
    struct per_cpu_pages *pcp;
    unsigned long flags;
    unsigned long pfn = page_to_pfn(page); // 根据给出页地址求出对应的页帧号
    int migratetype;
    // 释放前pcp的准备工作，检查释放满足释放条件
    if (!free_pcp_prepare(page))
        return;
    // 获取页框所在pageblock的页框类型
    migratetype = get_pfnblock_migratetype(page, pfn);
    // 设置页框类型为pageblock的页框类型，因为在页框使用过程中，这段pageblock可以移动
    set_pcpage_migratetype(page, migratetype);
    local_irq_save(flags);
    _count_vm_event(PGFREE);
    // 如果不是高速缓存类型，就放回到伙伴系统中
    if (migratetype >= MIGRATE_PCPTYPES) {
        if (unlikely(is_migrate_isolate(migratetype))) {
            free_one_page(zone, page, pfn, 0, migratetype);
            goto out;
        }
        migratetype = MIGRATE_MOVABLE;
    }
```

图 17: free_hot_cold_page 函数的源代码示意图

综上所述，以上函数的执行流程图如图 18 所示：

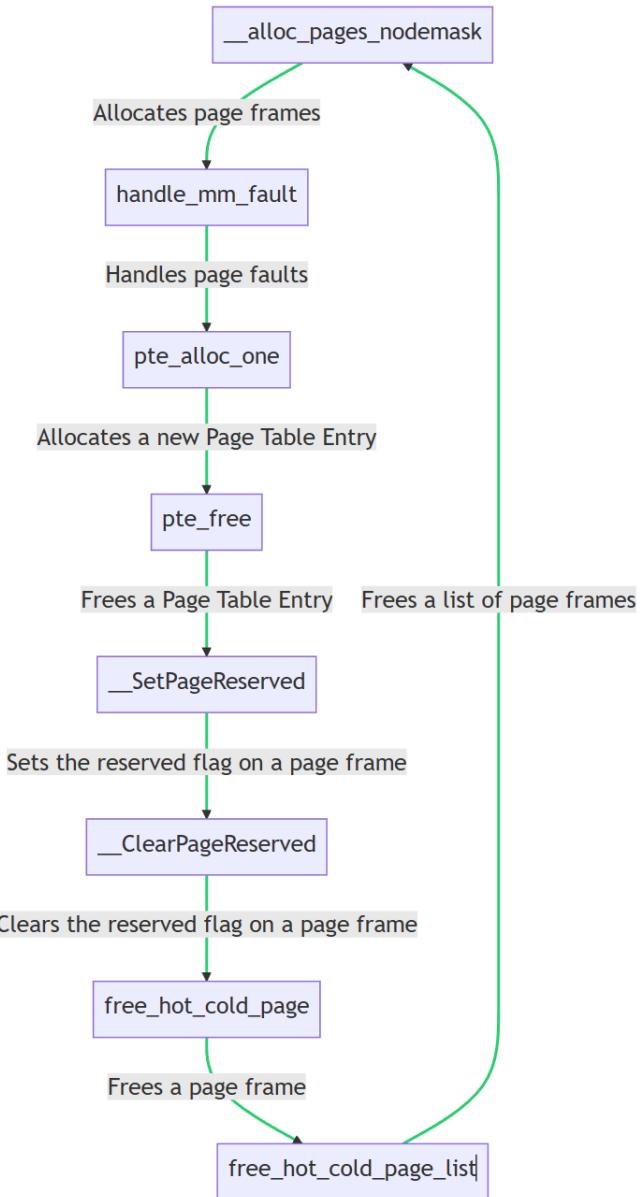


图 18：Linux 页式内存管理相关函数的执行流程图

首先，`__alloc_pages_nodemask` 函数分配页框。当发生缺页异常时，`handle_mm_fault` 函数会被调用。这可能会导致需要分配一个新的页表项，由 `pte_alloc_one` 函数完成。当不再需要一个页表项时，`pte_free` 函数会被调用来释放它。`__SetPageReserved` 和 `__ClearPageReserved` 函数用于设置和清除页框的保留标志。最后，`free_hot_cold_page` 和 `free_hot_cold_page_list` 函数用于释放页框，将其返回到页框分配器。

2.3 内核跟踪和调试程序设计

为了打印出特定应用 (HustSSE) 程序调入内存后占用了哪些页框，我在 struct page *__alloc_pages(gfp_t gfp, unsigned int order, int preferred_nid, nodemask_t *nodemask) 函数下增添了下列代码，用于打印相关信息，如图 19 所示：

```
5445 if(strcmp(current->comm, "HustSSE") == 0)
5446 {
5447     printk(KERN_INFO "HustSSE: Allocate %d page(s) at phy_addr: %llx, pfn:%llx, vir_addr: %l
5448         | 1 << order, page_to_phys(page), page_to_pfn(page), page_to_virt(page));
5449 }
5450 }
```

图 19：利用 printk 打印相关信息

同时，为了打印出页框被释放时的信息，我在 void __free_pages(struct page *page, unsigned int order) 函数下增添了下列代码，用于打印相关信息，如图 20 所示：

```
5497 void __free_pages(struct page *page, unsigned int order)
5498 {
5499     /* get PageHead before we drop reference */
5500     int head = PageHead(page);
5501     if(strcmp(current->comm, "HustSSE") == 0)
5502     {
5503         printk(KERN_INFO "HustSSE: Freed %d page(s) at %llx\n",
5504             | 1 << order, page_to_phys(page));
5505     }
5506     if (put_page_testzero(page))
5507     {
5508         free_the_page(page, order);
5509     }
5510     else if (!head)
5511         while (order-- > 0)
5512             free_the_page(page + (1 << order), order);
5513     }
5514 }
5515 EXPORT_SYMBOL(__free_pages);
```

图 20：利用 printk 打印相关信息

经过苏曙光老师的提醒，我意识到程序装入内存过程的复杂性，在他的提示下，我研究了 load_elf_binary 函数，如图 21 所示：

```

823 static int load_elf_binary(struct linux_binprm *bprm)
824 {
825     struct file *interpreter = NULL; /* to shut gcc up */
826     unsigned long load_addr, load_bias = 0, phdr_addr = 0;
827     int load_addr_set = 0;
828     unsigned long error;
829     struct elf_phdr *elf_ppnt, *elf_phdata, *interp_elf_phdata = NULL;
830     struct elf_phdr *elf_property_phdata = NULL;
831     unsigned long elf_bss, elf_brk;
832     int bss_prot = 0;
833     int retval, i;
834     unsigned long elf_entry;
835     unsigned long e_entry;
836     unsigned long interp_load_addr = 0;
837     unsigned long start_code, end_code, start_data, end_data;
838     unsigned long reloc_func_desc __maybe_unused = 0;
839     int executable_stack = EXSTACK_DEFAULT;
840     struct elfhdr *elf_ex = (struct elfhdr *)bprm->buf;
841     struct elfhdr *interp_elf_ex = NULL;
842     struct arch_elf_state arch_state = INIT_ARCH_ELF_STATE;
843     struct mm_struct *mm;
844     struct pt_regs *regs;
845
846     retval = -ENOEXEC;
847     /* First of all, some simple consistency checks */

```

图 21: load_elf_binary 函数

在程序装入内存的片段打印了如下的调试信息，如图 22 所示：

```

1203
1204     e_entry = elf_ex->e_entry + load_bias;
1205     phdr_addr += load_bias;
1206     elf_bss += load_bias;
1207     elf_brk += load_bias;
1208     start_code += load_bias;
1209     end_code += load_bias;
1210     start_data += load_bias;
1211     end_data += load_bias;
1212     if(strcmp(current->comm, "HustSSE") == 0)
1213     {
1214         printk(KERN_INFO "HustSSE: at load_binary_elf, Allocate page(s) at %lx , pfn:
1215         // printk(KERN_INFO "HustSSE: Allocate page(s) at %lld, pfn: %lld\n",page_to_p
1216

```

图 22：利用 printk 打印相关信息

2.4 运行和测试过程

首先，我编写了 HustSSE.c 的测试程序，编译后在 sudo 下运行，同时使用 dmesg -w 查看内核信息，如图 23 所示。

```

文件 编辑 选择 查看 转到 运行 终端 帮助
C HustSSE.c > main()
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4
5 #define SIZE 1024 * 1024 * 100 // 100MB
6
7 int main() {
8     char *buffer;
9
10    // 分配内存
11    buffer = (char *)malloc(SIZE);
12    if (!buffer) {
13        printf("Memory allocation failed!\n");
14        return -1;
15    }
16
17    printf("Memory allocated. Sleeping...\n");
18
19    // 暂停一段时间
20    sleep(10);
21
22    // 释放内存
23    free(buffer);
24
25    printf("Memory freed. Exiting...\n");
26

```

行 10, 列 12 空格:4 UTF-8 C Linux

图 23：编写测试程序

可以看到程序被装入时、程序运行期间分配的页框和运行结束后释放的页框，如图 24 所示。

```

活动 终端 > Player(P) || 活动
dekrt@dekrt-virtual-machine: ~/code/Task2$ dmesg -w
[21036.715992] HustSSE: Allocate 1 page(s) at phy_addr: 1a7703000, pfn:1a7703, vir_addr: ffff89aa27703000
[21036.716010] HustSSE: Allocate 1 page(s) at phy_addr: 16348f000, pfn:16348f, vir_addr: ffff89aa9e348f000
[21036.716041] HustSSE: at load binary_elf, Allocate page(s) at 555c46e480c0 , pfn: 555c46e480c0
[21036.716046] HustSSE: Allocate 1 page(s) at phy_addr: 14323d000, pfn:14323d, vir_addr: ffff89aa9c323d000
[21036.716048] HustSSE: Allocate 1 page(s) at phy_addr: 14323c000, pfn:14323c, vir_addr: ffff89aa9c323c000
[21036.716053] HustSSE: Allocate 1 page(s) at phy_addr: 1d86a8000, pfn:1d86a8, vir_addr: ffff89aa586a8000
[21036.716066] HustSSE: Allocate 1 page(s) at phy_addr: 216c19000, pfn:216c19, vir_addr: ffff89aa9e19000
[21036.716104] HustSSE: Allocate 1 page(s) at phy_addr: 112f5d000, pfn:112f5d, vir_addr: ffff89aa992f5d000
[21036.716107] HustSSE: Allocate 1 page(s) at phy_addr: 21b6ac000, pfn:21b6ac, vir_addr: ffff89aa9ebac000
[21036.716110] HustSSE: Allocate 1 page(s) at phy_addr: 1561d2000, pfn:1561d2, vir_addr: ffff89aa9d61d2000
[21036.716129] HustSSE: Allocate 1 page(s) at phy_addr: 18628c000, pfn:18628c, vir_addr: ffff89aa9a628c000
[21036.716180] HustSSE: Allocate 1 page(s) at phy_addr: 19786a000, pfn:19786a, vir_addr: ffff89aa17986a000
[21036.716206] HustSSE: Allocate 1 page(s) at phy_addr: 1ddbf1000, pfn:1ddbf1, vir_addr: ffff89aa5fdb1000
[21036.716274] HustSSE: Allocate 1 page(s) at phy_addr: 13eb08000, pfn:13eb08, vir_addr: ffff89aa9beb08000
[21036.716309] HustSSE: Allocate 1 page(s) at phy_addr: 182f33000, pfn:182f33, vir_addr: ffff89aa02f33000
[21036.716545] HustSSE: Allocate 1 page(s) at phy_addr: 216edb000, pfn:216edb, vir_addr: ffff89aa96edb000
[21036.716574] HustSSE: Allocate 1 page(s) at phy_addr: 182f32000, pfn:182f32, vir_addr: ffff89aa02f32000
[21036.716627] HustSSE: Allocate 1 page(s) at phy_addr: 14b5c0000, pfn:14b5c0, vir_addr: ffff89aa9cbc50000
[21036.716716] HustSSE: Allocate 1 page(s) at phy_addr: 1cc4c9000, pfn:1cc4c9, vir_addr: ffff89aa4a4c4c9000
[21036.716725] HustSSE: Allocate 1 page(s) at phy_addr: 216ed9000, pfn:216ed9, vir_addr: ffff89aa9a6ed9000
[21036.716762] HustSSE: Allocate 1 page(s) at phy_addr: 1caf93000, pfn:1caf93, vir_addr: ffff89aa4af93000
[21036.716769] HustSSE: Allocate 1 page(s) at phy_addr: 19a55c000, pfn:19a55c, vir_addr: ffff89aa1a15c000
[21036.716775] HustSSE: Allocate 1 page(s) at phy_addr: 1d9c6e000, pfn:1d9c6e, vir_addr: ffff89aa59c6e000
[21036.716792] HustSSE: Allocate 1 page(s) at phy_addr: 20f480000, pfn:20f480, vir_addr: ffff89aa8f480000
[21036.716800] HustSSE: Allocate 1 page(s) at phy_addr: 2011bc000, pfn:2011bc, vir_addr: ffff89aa811bc000
[21036.716933] HustSSE: Allocate 1 page(s) at phy_addr: 19f83b000, pfn:19f83b, vir_addr: ffff89aa1f83b000
[21036.717421] HustSSE: Freed 1 page(s) at 19f83b000
[21036.717441] HustSSE: Allocate 1 page(s) at phy_addr: 19ab71000, pfn:19ab71, vir_addr: ffff89aa1ab71000
[21036.717445] HustSSE: Allocate 1 page(s) at phy_addr: 1f6fef000, pfn:1f6fef, vir_addr: ffff89aa76fef000
[21036.717470] HustSSE: Allocate 1 page(s) at phy_addr: 1dfd40000, pfn:1dfd40, vir_addr: ffff89aa5fd40000
[21036.717599] HustSSE: Allocate 1 page(s) at phy_addr: 20de64000, pfn:20de64, vir_addr: ffff89aa8de64000
[21036.717608] HustSSE: Allocate 1 page(s) at phy_addr: 19f83b000, pfn:19f83b, vir_addr: ffff89aa1f83b000
[21036.717610] HustSSE: Allocate 1 page(s) at phy_addr: 1718a5000, pfn:1718a5, vir_addr: ffff89aa9f18a5000
[21036.718166] HustSSE: Allocate 1 page(s) at phy_addr: 19f83a000, pfn:19f83a, vir_addr: ffff89aa1f83a000
[21036.718169] HustSSE: Allocate 1 page(s) at phy_addr: 19f838000, pfn:19f838, vir_addr: ffff89aa1f838000
[21036.718172] HustSSE: Allocate 1 page(s) at phy_addr: 1fb843000, pfn:1fb843, vir_addr: ffff89aa7b43000
[21046.718634] HustSSE: Allocate 1 page(s) at phy_addr: 19f838000, pfn:19f838, vir_addr: ffff89aa1f838000
[21046.718742] HustSSE: Freed 1 page(s) at 19f838000

```

图 24：利用 dmesg 查看相关信息

2.5 实验心得和建议

1. **实验心得：**通过阅读和理解 Linux 内核源代码，我对 Linux 的内存管理有了更深入的理解。我了解到了页框分配、缺页处理、页表项的分配和释放、页框的保留标志设置和清除以及页框的释放等过程。这些过程都是由一系列精心设计的函数完成的，这些函数共同构成了 Linux 内核中的页面/页框管理模块。通过这次实验，我对这些函数以及它们之间的关系有了更清晰的认识。此外，我还学习了如何将这些函数的执行流程用图表的形式表示出来，这对于理解和记忆这些复杂的过程非常有帮助。

2. 遇到的问题和解决办法：

- **问题 1：**在阅读源代码时，我发现有些函数的功能并不是一目了然的。这给我理解代码带来了一些困难。
- **解决办法：**我查阅了相关的文档和资料，以及在网上搜索了相关的讨论和解释。这些资料帮助我理解了这些函数的功能和用途。
- **问题 2：**在绘制函数执行流程图时，我发现有些函数之间的关系并不是很明显，这使得流程图的绘制变得困难。
- **解决办法：**我仔细阅读了源代码，尝试理解每个函数的输入和输出，以及它们是如何互相调用的。这使我能够更准确地绘制出函数执行的流程图。
- **问题 3：**在编译内核之后，加载编译后的新内核时提示：Initrd is too big，无法加载新内核。
- **解决办法：**我仔细上网查找，发现使用虚拟机与双系统不同，需要在编译内核的 `make modules_install` 一步修改为：`make INSTALL_MOD_STRIP=1 modules_install`，在编译时不生成调试信息，以此减少 initrd 的大小。修改完

成后，新内核成功加载。

3. 建议：对于初次接触 Linux 内核源代码的人来说，这个实验可能会有些困难。我建议在开始实验之前，先对 Linux 的内存管理有一些基本的了解，例如了解什么是页框、页表等概念。此外，阅读源代码时，不要期望一下子就能理解所有的内容。有时候，需要反复阅读，甚至需要查阅其他的资料和文档，才能理解代码的含义。最后，我建议在阅读源代码时，尽量将代码的功能和实际的操作联系起来，这样可以更好地理解代码的作用。

2.6 学习和编程实现参考网址

1. **Linux 内核源代码**：我直接阅读了 Linux 5.15.111 的内核源代码，特别是与页面/页框管理相关的部分。源代码可以在以下网址找到：[Linux Kernel Source Code](#)
2. **Linux 内存管理**：这个网址提供了关于 Linux 内存管理的详细介绍，包括页框分配、缺页处理、页表项的分配和释放等过程。我在阅读源代码时，参考了这个网址的内容：[Linux Memory Management](#)
3. **Linux 内核源代码解析**：这个网址提供了对 Linux 内核源代码的详细解析，我在理解某些复杂的函数时，参考了这个网址的内容：[Linux Kernel Source Code Analysis](#)
4. **Mermaid**：这个网址提供了关于 Mermaid 的详细介绍，我在绘制函数执行流程图时，参考了这个网址的内容：[Mermaid Live Editor](#)