西南大学人工智能学院

《操作系统》课程

考试科目：	操作系统
教　　师：	贾文柱
学　　年：	2023
学　　期：	第一学期
年级专业：	2020 级智能科学与技术专业
学　　号：	222020335220192
姓　　名：	严中圣
成　　绩：	
评　　语：	

人工智能学院制

课程论文（或案例等）（共 3 题，共 100 分）

要求：

一、 学习 Linux 中 module 模块的编程基本方法，并可以将其动态加载入内核。（请将每一步的过程截图保存）

二、 线程信号量同步与互斥：生产者和消费者共享 1 个缓冲区，使用之前学过的方法创建两个线程分别作为生产者和消费者

三、 内存管理: 页面转换算法(屏幕上应该输出每一步的中间结果，不可以只有最后一个数值)

要求：

(1)使用数组存储一组页面请求，页面请求的数量要 50 个以上，访问的页面号可以用随机数生成（0~20）；

(2)设置为分配给进程的页框数（假定是 5）；

(3)使用 LRU 算法，模拟完成全部的页面请求，最后输出总共发生了多少次缺页;

(4) 重新设置页框为 10，重复第 2 和第 3 步，观察页框数量对缺页中断率的影响;

(5) 在相同页框的情况下，使用 FIFO 算法模拟全部的页面请求，以此来比对 FIFO 和 LRU 之间的差别。

格式要求：

字数不限，大标题采用黑体二号，段落小标题采用宋体三号，正文采用宋体小四号，单倍行距。

本页打印在封面背后，下一页正式开始论文，全篇采用 A4 纸双面打印，左侧装订。

题目一：Linux 内核编程方法

1. 实验原理

在 Linux 中，内核模块是一种可以加载和卸载的动态扩展，用于扩展和修改内核的功能。编写 Linux 内核模块涉及以下基本方法：

1) 准备开发环境：首先，确保你的系统已经安装了适当的开发工具，如 GCC 编译器和相关的开发包。此外，你需要内核源代码或内核头文件来编译你的模块。

2) 创建模块源代码文件：创建一个新的源代码文件，使用 C 语言编写你的模块代码。模块源代码文件的扩展名通常是.c。

3) 包含必要的头文件：在你的模块源代码文件中，包含必要的头文件，如 linux/module.h 和 linux/kernel.h。这些头文件包含了模块编程所需的函数和宏定义。

4) 定义模块初始化和清理函数：在你的模块源代码中，定义一个模块初始化函数和一个模块清理函数。模块初始化函数在加载模块时被调用，用于初始化模块。模块清理函数在卸载模块时被调用，用于清理模块所占用的资源。

5) 使用宏定义注册模块：在你的模块源代码中，使用合适的宏定义来注册你的模块。常用的宏定义包括 module_init 和 module_exit，它们用于指定模块的初始化和清理函数。

6) 编译模块：使用 GCC 编译你的模块源代码，生成一个模块文件。你可以使用 Makefile 来自动化编译过程。

7) 加载和卸载模块：使用 insmod 命令加载你的模块，使用 rmmod 命令卸载模块。加载模块将调用模块的初始化函数，卸载模块将调用模块的清理函数。

这是一个基本的 Linux 内核模块编程方法的概述。请注意，编写和加载内核模块可能涉及系统的特权操作，因此要小心操作并理解你的代码对系统的影响。在编写和加载模块之前，请确保你对操作系统和内核编程有一定的了解。

2. 实验过程

下面是一个案例，展示了如何编写一个简单的 Linux 内核模块，名为"hello"。该模块在内核加载时输出一条消息"Hello, world!"，并在内核卸载时输出一条消息"Goodbye, world!"。

首先编写符合 linux 格式的模块文件：

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

static int __init hello_init(void)
{
    printk(KERN_INFO "Hello, world!\\n");
    return 0;
}
static void __exit hello_exit(void)
{
```

```
    printk(KERN_INFO "Goodbye, world!\\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_DESCRIPTION("A simple hello world module");
```

其次编写 Makefile 文件：

```
obj-m := hello.o #要生成的模块名

KERNELDIR ?= /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)

all:
   make -C $(KERNELDIR) M=$(PWD) modules

clean:
   make -C $(KERNELDIR) M=$(PWD) clean
```

相关配置说明如下：

- CONFIG_MODULE_SIG=n：用来关闭内核模块签名验证。在某些版本的内核中，需要使用该选项来编译内核模块，否则会出现签名验证失败的问题。
- obj-m := hello.o：指定要生成的模块名为 hello.o，也就是要编译的源文件名为 hello.c。
- KERNELDIR ?= /lib/modules/$(shell uname -r)/build：指定内核源代码的路径。如果没有设置该变量，则默认使用/lib/modules/$(shell uname -r)/build 作为路径。其中，$(shell uname -r)表示获取当前系统内核的版本号。
- PWD := $(shell pwd)：将当前目录的路径保存到变量 PWD 中。
- all:：默认的目标。在执行 make 命令时，如果没有指定目标，则默认执行该目标下的所有命令。
- make -C $(KERNELDIR) M=$(PWD) modules：编译当前目录下的源代码，并生成一个名为 hello.ko 的内核模块文件。其中，C 选项指定内核源代码的路径，M 选项指定模块源代码的路径，modules 表示编译为一个内核模块文件。
- clean:：清理目标。在执行 make clean 命令时，会执行该目标下的所有命令。
- make -C $(KERNELDIR) M=$(PWD) clean：清理当前目录下的所有编译生成的文件。

然后在控制台执行以下代码：

```
- sudo make：编译当前目录下的代码，并生成可执行文件或者模块文件。
- ls：列出当前目录下的文件。
- sudo insmod hello.ko：将名为 hello.ko 的模块文件插入内核中。
- lsmod：列出当前已经插入内核的模块文件。
- modinfo hello.ko：查看名为 hello.ko 的模块文件的详细信息。
- dmesg：显示内核在启动时产生的所有日志消息，包括模块加载和卸载时的消息。
- sudo rmmod hello：将名为 hello 的模块文件从内核中移除。
```

以下为结果图：

```
make -C /lib/modules/5.15.0-58-generic/build M=/home/xiaoge4/OS-HW-ALL/OS-HW-1 modules
make[1]: Entering directory '/usr/src/linux-headers-5.15.0-58-generic'
  CC [M]  /home/xiaoge4/OS-HW-ALL/OS-HW-1/hello.o
  MODPOST /home/xiaoge4/OS-HW-ALL/OS-HW-1/Module.symvers
  CC [M]  /home/xiaoge4/OS-HW-ALL/OS-HW-1/hello.mod.o
  LD [M]  /home/xiaoge4/OS-HW-ALL/OS-HW-1/hello.ko
  BTF [M] /home/xiaoge4/OS-HW-ALL/OS-HW-1/hello.ko
Skipping BTF generation for /home/xiaoge4/OS-HW-ALL/OS-HW-1/hello.ko due to unavailability of vmlinu
x
make[1]: Leaving directory '/usr/src/linux-headers-5.15.0-58-generic'
            :                  $ ls
hello.c   hello.mod    hello.mod.o  Makefile       Module.symvers
hello.ko  hello.mod.c  hello.o      modules.order
            :                  $ sudo insmod hello.ko
            :                  $ dmesg
             Linux version 5.15.0-58-generic (buildd@lcy02-amd64-033) (gcc (Ubuntu 9.4.0-1ubuntu1~
20.04.1) 9.4.0, GNU ld (GNU Binutils for Ubuntu) 2.34) #64~20.04.1-Ubuntu SMP Fri Jan 6 16:42:31 UTC
 2023 (Ubuntu 5.15.0-58.64~20.04.1-generic 5.15.74)
```

```
ernel
              Hello, world!\n
              Goodbye, world!\n
              Hello, world!\n
              Goodbye, world!\n
              Hello, world.Sheng\n
              Goodbye, world.Sheng\n
            :                  $ lsmod
Module                  Size  Used by
hello                  16384  0
exfat                  86016  1
uas                    28672  0
usb_storage            77824  2 uas
rfcomm                 81920  4
ccm                    20480  6
cmac                   16384  3
algif_hash             16384  1
algif_skcipher         16384  1
af_alg                 32768  6 algif_hash,algif_skcipher
bnep                   28672  2
intel_rapl_msr         20480  0
nls_iso8859_1          16384  1
```

题目二：线程信号量同步与互斥

1. 实验原理

线程信号量同步与互斥是一种常用的线程间通信和同步机制。在生产者-消费者问题中，生产者线程负责向缓冲区中生产数据，消费者线程负责从缓冲区中消费数据。通过使用信号量来控制缓冲区的访问，可以实现生产者和消费者之间的同步和互斥操作，确保数据的正确性和顺序性。

以下是使用线程信号量实现生产者-消费者问题的基本步骤：

1) 创建一个共享的缓冲区，用于生产者和消费者之间的数据交换。
2) 定义两个信号量：
   - 一个用于表示可用的数据数量，初始值为 0，称为"空槽位"信号量。
   - 一个用于表示空闲的槽位数量，初始值为缓冲区的大小，称为"数据可用"信号量。
3) 创建生产者线程：
   - 生产者线程循环执行以下步骤：
     - 等待"空槽位"信号量，如果缓冲区没有空槽位，则等待。
     - 向缓冲区中写入数据。
     - 递增"数据可用"信号量，表示有新的数据可用。
4) 创建消费者线程：
   - 消费者线程循环执行以下步骤：
     - 等待"数据可用"信号量，如果缓冲区没有可用的数据，则等待。
     - 从缓冲区中读取数据。

- 递增"空槽位"信号量，表示释放了一个空槽位。

5) 启动生产者和消费者线程：
   - 创建并启动生产者线程。
   - 创建并启动消费者线程。

通过上述步骤，生产者和消费者线程将交替执行，并通过信号量来实现互斥和同步操作。当缓冲区已满时，生产者线程将等待，直到有可用的槽位。当缓冲区为空时，消费者线程将等待，直到有可用的数据。这样，生产者和消费者线程之间就能够正确地进行数据交换，保证了数据的完整性和顺序性。

2. 实验过程

首先编写生产者和消费者对应线程：

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

#define BUFFER_SIZE 10

int buffer[BUFFER_SIZE];
int in = 0;
int out = 0;

sem_t mutex;
sem_t empty;
sem_t full;

int produce_item() {
   return 1;
}

void consume_item(int item) {
   printf("Consumed item %d\\n", item);
}

void *producer(void *arg) {
   while (1) {
       int item = produce_item();
       printf("Produced item %d\\n", item);
       sem_wait(&empty);
       sem_wait(&mutex);
       buffer[in] = item;
       in = (in + 1) % BUFFER_SIZE;
       sem_post(&mutex);
       sem_post(&full);
```

```
    }
}

void *consumer(void *arg) {
    while (1) {
        sem_wait(&full);
        sem_wait(&mutex);
        int item = buffer[out];
        printf("Consumed item %d\\n", item);
        out = (out + 1) % BUFFER_SIZE; //模拟环形缓冲区
        sem_post(&mutex);
        sem_post(&empty);
        consume_item(item);
    }
}

int main() {
    sem_init(&mutex, 0, 1);
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);

    pthread_t producer_thread, consumer_thread;
    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_create(&consumer_thread, NULL, consumer, NULL);

    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);

    sem_destroy(&mutex);
    sem_destroy(&empty);
    sem_destroy(&full);

    return 0;
}
```

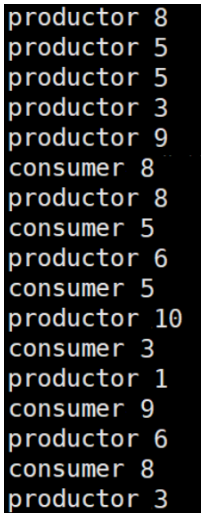
其中：

- 定义了一个大小为 BUFFER_SIZE 的缓冲区数组 buffer，用于存放生产者生产的产品；
- 定义了两个变量 in 和 out，分别表示生产者放置产品的位置和消费者取产品的位置，模拟一个环形缓冲区；
- 定义了三个信号量 mutex、empty 和 full，分别用于互斥访问缓冲区、表示空闲缓冲区的数量和表示产品数量的数量；
- 定义了 producer 函数和 consumer 函数，分别表示生产者线程和消费者线程的执行函数；
- 在 main 函数中，初始化信号量，创建生产者线程和消费者线程，并等待两个线程结束后销毁信号量。

在终端执行以下代码结果如下：

- `gcc -pthread -o producer_consumer producer_consumer.c` 编译
- `./producer_consumer` 运行可执行文件

可在终端中观察生产者和消费者线程的输出，以及缓冲区的状态。



题目三：页面转换

1. 实验原理

页面转换算法用于管理虚拟内存和物理内存之间的映射关系，以满足进程对内存的访问需求。在本实验中，我们使用 LRU 算法和 FIFO 算法来模拟页面转换的过程，并观察不同页框数量下的缺页中断率。

LRU 算法基于页面的访问历史，认为最近被访问的页面是最有可能再次被访问的。因此，LRU 算法选择最近最少被使用的页面进行置换。FIFO 算法基于页面的进入顺序，认为最早进入的页面是最有可能被置换出去的。因此，FIFO 算法选择最早进入的页面进行置换。

2. 实验过程

1) 使用数组存储一组页面请求，页面请求的数量要 50 个以上，访问的页面号可以用随机数生成（0~20）；

```
const int PAGE_NUM = 50;
int page_req[PAGE_NUM];
srand(time(NULL));

// generate random page requests
for (int i = 0; i < PAGE_NUM; i++) {
   page_req[i] = rand() % 20;
}
```

2) 设置为分配给进程的页框数（设定为 5）；

```
const int PAGE_FRAME_NUM = 5;
int page_frame[PAGE_FRAME_NUM];
```

3) 使用 LRU 算法。

首先，我们初始化一个页框数组和缺页中断次数为 0。然后，对于每个页面请求，我们检查当前请求的页面是否已经在页框数组中。如果页面已经存在，我们将该页面移动到数组的末尾，表示该页面是最近被使用的。如果页面不在页框数组中，意味着发生了缺页中断。在这种情况下，我们需要进行页面置换。如果页框数组已满，我们选择最近最少被使用的页面（即页框数组中的第一个页面），将其替换为新的页面。如果页框数组未满，我们直接将新的页面添加到页框数组的末尾。在每次页面请求后，我们输出当前页框数组的内容，以便观察页面置换的过程。最后，我们输出缺页中断次数，表示整个过程中发生的缺页情况。

```
int page_fault = 0;
int found;

// simulate LRU
for (int i = 0; i < PAGE_NUM; i++)
{
    found = 0;
    // check if page is already in frame
    for (int j = 0; j < PAGE_FRAME_NUM; j++)
    {
        if (page_frame[j] == page_req[i])
        {
            found = 1;
            break;
        }
    }
    // if page is not in frame, find the oldest page and replace it
    if (!found)
    {
        page_fault++;
        printf("Page fault at index %d\n", i);
        for (int k = 0; k < FRAME_NUM; k++)
        { // 输出页框数组中的内容
            printf("%d ", frames[k]);
        }
        printf("\n");
        if (i < PAGE_FRAME_NUM)
        {
            page_frame[i] = page_req[i];
        }
        else
```

```
        {
            int oldest_page = 0;
            int oldest_time = PAGE_NUM;
            for (int j = 0; j < PAGE_FRAME_NUM; j++)
            {
                for (int k = i - 1; k >= 0; k--)
                {
                    if (page_frame[j] == page_req[k])
                    {
                        if (k < oldest_time)
                        {
                            oldest_page = j;
                            oldest_time = k;
                        }
                        break;
                    }
                }
                if (k < 0)
                {
                    oldest_page = j;
                    break;
                }
            }
            page_frame[oldest_page] = page_req[i];
        }
    }
}
```

实验结果如下：

```
Page fault at index 39
17 19 16 18 10
Page fault at index 41
17 19 16 18 14
Page fault at index 42
17 19 16 18 12
Page fault at index 45
17 19 16 18 1
Page fault at index 46
17 19 16 18 15
Page fault at index 47
17 19 16 18 11
Page fault at index 48
17 19 16 18 1
Page fault at index 49
17 19 16 18 0
Total page faults: 40
```

在这个实验中，我们观察了 50 个随机页面请求的情况，一共发生了 40 次缺页中断。通过输出页面请求和页框数组的变化情况，以及使用 LRU 算法进行页面替换的过程，我们可以清晰地观察到 LRU 算法的实现过程。

开始时，所有的页框都是空的，页框数组中的值都为-1。每次有新的页面请求到达时，页框数组会更新，并打印当前的页框数组状态。如果所有页框都被占满，而又有新的页面请求到达，则使用 LRU 算法选择最近最久未使用的页面进行替换。在输出结果中，我们可以观察到每次替换的页面编号以及替换后的页框数组状态。

通过观察输出结果，我们可以清楚地了解 LRU 算法的实现过程。当新的页面请求到达时，算法检查页框数组是否已经包含该页面。如果存在，则页面已经在内存中，无需替换。如果不存在，则需要选择合适的页面进行替换，LRU 算法选择最近最久未使用的页面进行替换，以为新页面腾出空间。

4) 重新设置页框为 10，重复第 2 和第 3 步，观察页框数量对缺页中断率的影响；

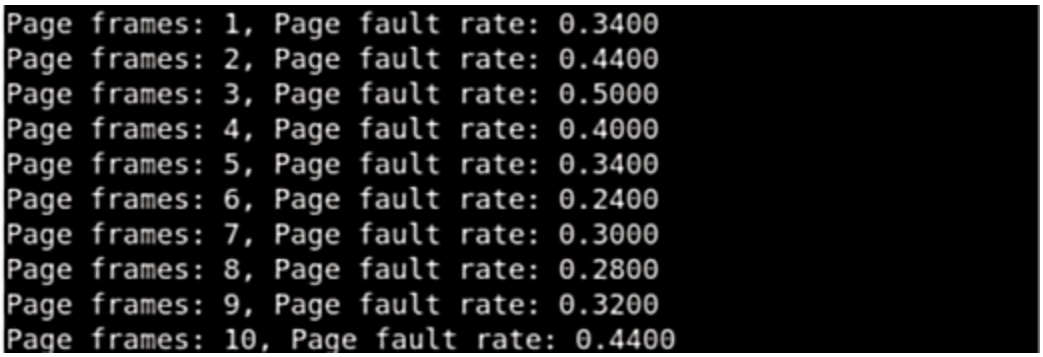
```
const int NEW_PAGE_FRAME_NUM = 10;
int new_page_frame[NEW_PAGE_FRAME_NUM];
int new_page_fault = 0;

// simulate LRU with more page frames
for (int i = 0; i < NEW_PAGE_FRAME_NUM; i++) {
   new_page_frame[i] = -1;
}

for (int i = 0; i < PAGE_NUM; i++) {
   found = 0;
   // check if page is already in frame
   for (int j = 0; j < NEW_PAGE_FRAME_NUM; j++) {
       if (new_page_frame[j] == page_req[i]) {
           found = 1;
           break;
       }
  }
   // if page is not in frame, find the oldest page and replace it
   if (!found) {
       new_page_fault++;
       if (i < NEW_PAGE_FRAME_NUM) {
           new_page_frame[i] = page_req[i];
       } else {
           int oldest_page = 0;
           int oldest_time = PAGE_NUM;
           for (int j = 0; j < NEW_PAGE_FRAME_NUM; j++) {
               for (int k = i - 1; k >= 0; k--) {
                   if (new_page_frame[j] == page_req[k]) {
                       if (k < oldest_time) {
```

```
                        oldest_page = j;
                        oldest_time = k;
                    }
                     break;
                }
            }
             if (k < 0) {
                 oldest_page = j;
                 break;
            }
        }
         new_page_frame[oldest_page] = page_req[i];
    }
  }
}
float page_fault_rate = (float) new_page_faults / PAGE_NUM;
printf("Page frames: %d, Page fault rate: %.4f\n", frames,
page_fault_rate);
```

实验结果如下：



可以发现当页框数量较少时，由于内存空间有限，频繁进行页面置换，导致缺页中断率较高。随着页框数量增大，可用内存空间增加，缺页中断率显著下降，因为更多页面可以同时存在于内存中，减少了页面置换的次数。然而，一旦页框数量增加到 8 以上，进一步增加页框数量对缺页中断率的影响变得不明显，因为内存空间已经足够大，无法进一步显著降低缺页中断率。

在实际应用中，我们需要根据系统的内存资源和性能需求合理设置页框数量。如果页框数量太少，会导致频繁的页面置换和高缺页中断率，降低系统性能。如果页框数量太多，会占用过多的内存资源，造成资源浪费。因此，需要根据具体情况进行权衡和设置，以达到最佳的性能和资源利用效率。

5) 相同页框下，使用 FIFO 算法模拟全部的页面请求，并比对 FIFO 和 LRU 之间的差别。

首先初始化一个页框数组和缺页中断次数为 0。对于每个页面请求，我们检查当前请求的页面是否已经在页框数组中。如果页面已经存在，我们继续处理下一个请求。如果页面不在页框数组中，表示发生了缺页中断。在这种情况下，我们需要进行页面置换。如果页框数组已满，我们选择最早进入的页面（即页框数组中的第一个页面），将其替换为新

的页面。如果页框数组未满，我们直接将新的页面添加到页框数组的末尾。在每次页面请求后，我们输出当前页框数组的内容，以便观察页面置换的过程。最后，我们输出缺页中断次数，表示整个过程中发生的缺页情况。

```
int add_page_to_frames_fifo(int page_num, int frame_num, int* head,
int* tail) {
    int replaced_page = -1;
    if (get_page_frame_index(page_num) == -1) { // 页面不在内存中，需要替
换
        if (page_frames[*tail] != -1) { // 页框数组已满，需要替换
            replaced_page = page_frames[*tail];
        }
        page_frames[*tail] = page_num;
        *tail = (*tail + 1) % frame_num;
        if (*tail == *head) { // FIFO 算法的页面置换策略
            *head = (*head + 1) % frame_num;
        }
    }
    return replaced_page;
}
int main() {
    srand(time(NULL));
    for (int i = 0; i < PAGE_NUM; i++) {
        page_requests[i] = rand() % 20;
    }
    for (int frame_num = 1; frame_num <= MAX_FRAME_NUM; frame_num++) {
        for (int j = 0; j < MAX_FRAME_NUM; j++){
            page_frames[j] = -1;
        }
    int page_faults_lru = count_page_faults_lru(frame_num);
    float page_fault_rate_lru = (float) page_faults_lru / PAGE_NUM;
    printf("LRU: Page frames: %d, Total page faults: %d, Page fault
rate: %.4f\n", frame_num, page_faults_lru, page_fault_rate_lru);
    int page_faults_fifo = count_page_faults_fifo(frame_num);
    float page_fault_rate_fifo = (float) page_faults_fifo / PAGE_NUM;
    printf("FIFO: Page frames: %d, Total page faults: %d, Page fault
rate: %.4f\n", frame_num, page_faults_fifo, page_fault_rate_fifo);
    }
    return 0;
}
```

实验结果如下：

观察输出结果可以发现，随着页框数量的增加，缺页中断率逐渐下降。当页框数量增加到一定程度后，缺页中断率的下降趋势变得平缓。这是因为增加页框数量会提供更多的内存空间，减少了页面置换的频率，从而降低了缺页中断率。

此外，我们也可以观察到 LRU 算法的缺页中断率低于 FIFO 算法。这是因为 LRU 算法优先淘汰最近最少使用的页面，更加有效地利用了内存空间。相比之下，FIFO 算法只按照页面请求的先后顺序进行页面置换，没有考虑页面的访问频率。因此，在实际应用中，如果需要更好的页面置换算法，可以选择 LRU 算法来降低缺页中断率。