**操作系统课程设计实验报告**

# 1 基础题目

## 1.1 Shell编程

### 1.1.1 Shell脚本生成实验

*i. Shell脚本代码每句的含义*

# 检查传入的参数数量是否等于2  
if [ "$#" -ne 2 ]; then  
 # 如果参数数量不等于2，输出使用说明，并退出脚本  
 echo "Usage: $0 <filename> <r|w>"  
 exit 1  
fi  
  
# 提取传入的第一个参数，作为文件名  
filename="$1"  
# 提取传入的第二个参数，作为操作标志（r或w）  
flag="$2"  
  
# 初始化一个变量，用于存储生成的脚本内容  
script\_content="#!/bin/bash\n\n"  
script\_content+="filename=\"$filename\"\n"  
script\_content+="flag=\"$flag\"\n\n"  
script\_content+="if [ \"\$flag\" == \"r\" ]; then\n"  
# 如果操作标志是'r'，则读取并显示文件内容  
script\_content+=" cat \"\$filename\"\n"  
# 如果操作标志是'w'，则写入固定内容到文件  
script\_content+="elif [ \"\$flag\" == \"w\" ]; then\n"  
script\_content+=" echo \"472 MYFILE\" >| \"\$filename\"\n"  
# 如果操作标志既不是'r'也不是'w'，输出错误信息  
script\_content+="else\n"  
script\_content+=" echo \"无效参数. 请输入 'r' 或者 'w'.\"\n"  
script\_content+="fi\n"  
  
# 将生成的脚本内容写入一个新的脚本文件 script.sh  
echo -e "$script\_content" > script.sh  
  
# 修改生成的脚本文件的权限  
chmod +x script.sh  
  
# 执行生成的脚本文件  
./script.sh  
  
# 使用 strace 工具追踪生成脚本的 write 系统调用  
strace -e trace=write ./script.sh  
  
echo "生成的脚本被保存为 'script.sh' 并且执行了"

*ii. 自己通过哪些命令运行Shell脚本，这些命令的含义是什么？*

*bash T1.sh a.txt w 生成 script.sh 写 a.txt*

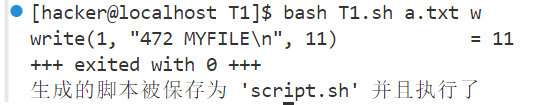
*bash T1.sh a.txt r 生成 script.sh 读 a.txt*

*iii. Bash解释执行自己编写的shell脚本背后的关键流程是什么*

1. **读取**：Bash读取脚本文件内容，逐行解释执行。
2. **解析**：Bash将每一行命令解析为内部命令、外部命令或内置函数。
3. **执行**：按顺序执行每个命令，并根据命令的执行结果决定后续操作。
4. **返回结果**：每个命令执行后返回一个状态码，Bash根据状态码决定是否继续执行后续命令或处理错误。
5. **变量替换**：在解析和执行过程中，Bash替换所有变量和命令替换的内容。
6. **控制流**：Bash根据脚本中的控制流结构（如if-else、for、while等）执行相应的分支和循环。

*iv.自己的 Shell脚本是否调用了write系统调用以及相关证据。*

是, 证据见截图



## 1.2 系统调用编程

### 1.2.1 Semaphore使用实验

*i. 自己程序中关键句的含义*

// 信号量（Semaphore）是一种用于进程间或线程间同步的机制。  
// 它是一个简单的计数器，用于控制对共享资源的访问。信号量可以用于解决资源竞争和实现互斥、同步等操作。  
  
#include <iostream>  
#include <fstream>  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <fcntl.h>  
#include <semaphore.h>  
#include <unistd.h>  
#include <sys/mman.h>  
#include <sys/wait.h>  
#include <cstring>  
  
const char sem\_name[] = "/my\_semaphore"; // 信号量名称  
const char file\_name[] = "output.txt"; // 文件名称  
  
void write\_to\_file(const char content[])  
{  
 std::ofstream file;  
 file.open(file\_name, std::ios::app);  
 if (file.is\_open())  
 {  
 file << content << std::endl;  
 file.close();  
 }  
 else  
 std::cerr << "无法打开文件" << std::endl;  
}  
  
int main()  
{  
 // 创建或打开信号量  
 // sem\_name 是信号量的名字，是一个以斜杠（/）开头的字符串。  
 // O\_CREAT 表示如果信号量不存在，则创建它。如果信号量已经存在，则直接打开它。  
 // 0644 表示信号量的权限，6 表示所有者的权限（可读和可写）, 4 表示组的权限（只读）(什么是组?), 4 表示其他用户的权限（只读）。  
 // 1 是信号量的初始值。信号量初始化为 1，表示当前资源是可用的。  
 sem\_t \*sem = sem\_open(sem\_name, O\_CREAT, 0644, 1);  
 if (sem == SEM\_FAILED)  
 {  
 perror("sem\_open");  
 exit(EXIT\_FAILURE);  
 }  
  
 // 创建第一个子进程  
 pid\_t pid1 = fork();  
  
 if (pid1 < 0)  
 {  
 perror("fork");  
 exit(EXIT\_FAILURE);  
 }  
 else if (pid1 == 0) // 子进程（进程1）  
 {  
 // sem\_wait 函数会尝试对信号量 sem 进行减 1 操作。  
 // 如果信号量的值大于 0，则将其值减 1 并立即返回，表示当前调用的进程或线程已成功获取到信号量，可以进入临界区。  
 // 如果信号量的值等于 0，则该调用进程将被阻塞，直到信号量的值大于 0 为止  
 sem\_wait(sem); // 获取信号量  
 write\_to\_file("472PROC1 MYFILE1"); // 写入文件  
 sem\_post(sem); // 释放信号量  
  
 // 关闭信号量  
 sem\_close(sem);  
 // 关闭进程 1  
 exit(EXIT\_SUCCESS);  
 }  
  
 // 父进程中 pid1 > 0 继续执行后续代码  
 // 创建第二个子进程  
 pid\_t pid2 = fork();  
  
 if (pid2 < 0)  
 {  
 perror("fork");  
 exit(EXIT\_FAILURE);  
 }  
 else if (pid2 == 0)  
 {  
 // 第二个子进程（进程2）  
 sem\_wait(sem); // 获取信号量  
 write\_to\_file("472PROC2 MYFILE2"); // 写入文件  
 sem\_post(sem); // 释放信号量  
  
 // 关闭信号量  
 sem\_close(sem);  
 exit(EXIT\_SUCCESS);  
 }  
  
 // 父进程等待两个子进程结束  
 waitpid(pid1, NULL, 0);  
 waitpid(pid2, NULL, 0);  
 sem\_close(sem); // 关闭信号量  
 sem\_unlink(sem\_name); // 删除信号量  
  
 return 0;  
}

#include <iostream>  
#include <fstream>  
#include <unistd.h>  
#include <sys/types.h>  
#include <sys/wait.h>  
#include <sys/ipc.h>  
#include <sys/shm.h>  
#include <stdlib.h>  
  
#define FILENAME "output.txt"  
  
int \*turn;  
  
void write\_to\_file(const std::string &content)  
{  
 std::ofstream file(FILENAME, std::ios::app);  
 if (!file.is\_open())  
 {  
 std::cerr << "Error opening file" << std::endl;  
 exit(EXIT\_FAILURE);  
 }  
 file << content << std::endl;  
 file.close();  
}  
  
void process\_task(const std::string &content, int my\_turn)  
{  
 for (int i = 0; i < 2; ++i)  
 {  
 // 等待 turn 变为 my\_turn  
 while (\*turn != my\_turn); // busy-waiting  
  
 // 进入临界区  
 write\_to\_file(content);  
  
 // 退出临界区，切换 turn  
 \*turn = 1 - my\_turn;  
  
 // 模拟非临界区操作  
 sleep(1);  
 }  
}  
  
int main()  
{  
 // 生成一个唯一的键值  
 key\_t key = ftok("output.txt", 65);  
 if (key == -1) {  
 perror("ftok");  
 exit(EXIT\_FAILURE);  
 }  
  
 // 创建共享内存段  
 int shmid = shmget(key, sizeof(int), 0666 | IPC\_CREAT);  
 if (shmid == -1) {  
 perror("shmget");  
 exit(EXIT\_FAILURE);  
 }  
  
 // 将共享内存段附加到进程的地址空间  
 turn = (int\*)shmat(shmid, (void\*)0, 0);  
 if (turn == (int\*)-1) {  
 perror("shmat");  
 exit(EXIT\_FAILURE);  
 }  
  
 \*turn = 1; // 初始化 turn  
  
 // 创建第一个进程  
 pid\_t pid1 = fork();  
 if (pid1 < 0)  
 {  
 perror("fork");  
 exit(EXIT\_FAILURE);  
 }  
 else if (pid1 == 0)  
 {  
 // 第一个子进程  
 process\_task("472PROC1 MYFILE1", 0);  
 exit(EXIT\_SUCCESS);  
 }  
  
 // 创建第二个进程  
 pid\_t pid2 = fork();  
 if (pid2 < 0)  
 {  
 perror("fork");  
 exit(EXIT\_FAILURE);  
 }  
 else if (pid2 == 0)  
 {  
 // 第二个子进程  
 process\_task("472PROC2 MYFILE2", 1);  
 exit(EXIT\_SUCCESS);  
 }  
  
 // 等待子进程结束   
 // NULL 表示不关心子进程的终止状态  
 // 0 表示使用默认行为，即等待子进程终止。  
 waitpid(pid1, NULL, 0);  
 waitpid(pid2, NULL, 0);  
  
 // 分离共享内存段  
 if (shmdt((void\*)turn) == -1) {  
 perror("shmdt");  
 exit(EXIT\_FAILURE);  
 }  
  
 // 删除共享内存段  
 if (shmctl(shmid, IPC\_RMID, NULL) == -1) {  
 perror("shmctl");  
 exit(EXIT\_FAILURE);  
 }  
  
 return 0;  
}

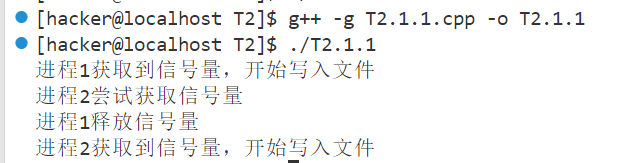
*ii. 请用实际操作证明当进程A占用semaphore后，进程B想要占用semaphore时，进程B进入睡眠。*

*可以简单地修改一下代码, 增加一些输出来证明*

else if (pid1 == 0)  
 {  
 sem\_wait(sem); // 获取信号量  
 std::cout << "进程1获取到信号量，开始写入文件" << std::endl;  
 write\_to\_file("472PROC1 MYFILE1"); // 写入文件  
 sleep(5); // 模拟进程1占用信号量一段时间  
 std::cout << "进程1释放信号量" << std::endl;  
 sem\_post(sem); // 释放信号量  
  
 sem\_close(sem);  
 exit(EXIT\_SUCCESS);  
 }  
  
 else if (pid2 == 0)  
 {  
 sleep(1); // 确保进程1先获取到信号量  
 std::cout << "进程2尝试获取信号量" << std::endl;  
 sem\_wait(sem); // 获取信号量  
 std::cout << "进程2获取到信号量，开始写入文件" << std::endl;  
 write\_to\_file("472PROC2 MYFILE2"); // 写入文件  
 sem\_post(sem); // 释放信号量  
  
 sem\_close(sem);  
 exit(EXIT\_SUCCESS);  
 }

在这段代码中，进程1在获取信号量后会睡眠5秒，这样进程2在尝试获取信号量时必然会被阻塞，直到进程1释放信号量。

输出结果为



*iii.移植课本中strict alternation算法时，该算法中的turn变量访问时是否需要加锁，以避免读写冲突？请回答并给出理由。*

不需要锁来保护 turn

1. turn 的读取和写入都是单一的原子操作, 不会满足竞争条件
2. 通过忙等待机制，两个进程能交替进入临界区

# 2 中级题目

## 2.1 进程调度管理

### 2.1.1 Runqueue长度观测实验

*i.ebpf程序执行的时机*

kprobe:enqueue\_task\_fair在内核函数**enqueue\_task\_fair**的入口处触发。**enqueue\_task\_fair**函数在任务被加入CFS运行队列时调用。

kprobe:dequeue\_task\_fair在内核函数**dequeue\_task\_fair**的入口处触发。**dequeue\_task\_fair**函数在任务被移出CFS运行队列时调用。

*ii. 程序中关键语句的含义。*

// kprobe 用于在 enqueue\_task\_fair 函数入口处插入探针, 在内核函数入口处触发。  
kprobe:enqueue\_task\_fair  
{  
 // 读取 cfs\_rq 结构并更新运行队列长度, arg0 是 enqueue\_task\_fair 的第一个参数 struct rq \*  
 @run\_queue\_length[cpu] = ((struct rq \*)arg0)->cfs.nr\_running  
}  
  
// 在任务被移出 CFS 运行队列时捕获 nr\_running  
kprobe:dequeue\_task\_fair  
{  
 // 读取 cfs\_rq 结构并更新运行队列长度  
 @run\_queue\_length[cpu] = ((struct rq \*)arg0)->cfs.nr\_running  
}  
  
// hz: 赫兹, 按照 99 赫兹的频率输出每个 cpu 的运行队列中的进程数  
profile:hz:99  
{  
 printf("CPU %d 运行队列中的进程数: %u\n", cpu, @run\_queue\_length[cpu]);  
}

### 2.1.2 进程切换次数观测实验

*i.ebpf程序执行的时机*

kprobe:dequeue\_task\_fair在内核函数**dequeue\_task\_fair**的入口处触发。当目标的进程退出运行队列时, 增加切换计数

*ii. 程序中关键语句的含义。*

// 在目标退出运行队列时增加切换计数  
kprobe:dequeue\_task\_fair  
{  
 if (pid == @pid) {  
 @switchcnt++; // 如果当前调度的进程ID与指定的PID相同，则增加切换计数  
 }  
}

### 2.1.3 RR调度算法定制

很遗憾本题代码实现未能完成目标, 存在问题

## 2.2 任选题（二选一）

*i. 自己所添加的系统调用的位置和修改点，以及为什么在这些位置上进行修改。*

*修改位置同高级题的第二题, 只是修改了系统调用名和调用号*

// 将新的系统调用添加到系统调用表中  
549 common write\_mynumber sys\_write\_mynumber

// 声明新的系统调用函数  
asmlinkage int sys\_write\_mynumber(int number, int flag);

// 定义系统调用号和映射  
#define \_\_NR\_write\_mynumber 549  
\_\_SYSCALL(\_\_NR\_write\_mynumber, sys\_write\_mynumber)

*ii.自己系统调用关键语句的含义。*

SYSCALL\_DEFINE2(write\_mynumber, int, number, int, flag)  
{  
 int ret = 0;  
 static DEFINE\_MUTEX(mynumber\_mutex);  
 printk("write mynumber is running!\n");  
 // 获取互斥锁  
 mutex\_lock(&mynumber\_mutex);  
  
 // 实现题目要求  
 if(flag) {  
 number /= 10;  
 ret = number % 10;  
 } else {  
 ret = number % 10;  
 }  
  
 // 释放互斥锁  
 mutex\_unlock(&mynumber\_mutex);  
  
 return ret;  
}

*Iii. 如何编译内核并调用自己的系统调用。*

编译内核命令如下:

sudo make all

sudo make modules

调用方法:

syscall(549, number, flag);

# 3 高级题目

## 3.1 内存管理

### 3.1.1 进程地址空间布局

*i.ebpf程序执行的时机*

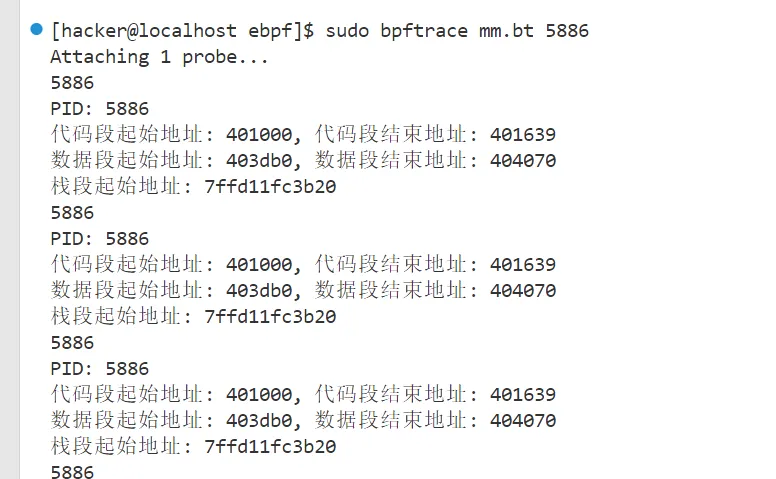
*当目前执行的进程是目标进程时触发*

*/ pid == (uint64)$1 /*

*ii. 程序中关键语句的含义*

// 获取当前任务结构  
$task = (struct task\_struct \*)curtask;   
// 获取内存标识符  
$mm = $task->mm;

*iii. 对输出的结果进行说明，特别需要指出输出的起始地址是物理地址还是虚拟地址。*



如输出所示, 输出结果为目标进程的代码段起始地址, 数据段起始地址. 栈起始地址

这些地址都是虚拟地址，因为进程在用户态运行时，操作系统提供给它的地址空间是虚拟的。虚拟地址需要通过内存管理单元（MMU）进行转换，映射到实际的物理内存地址。

在高级题的第二题中, 会展示如何通过虚拟地址找到物理地址

最后, 注: 由于 10 hz 太慢经常很久没有输出, 所以我将其改成了 999 hz

### 3.1.2 进程页表修改

*i.自己所添加的代码的逻辑*

*按照操作系统课程第四章 存储管理 的内容, 一步一步由虚拟地址找到物理地址, 然后将物理页面映射到内核地址空间并复制到用户空间地址即可*

*这里特别写了一个尝试获取页面的引用计数的代码, 用于防止在有其他的进程修改目标进程的页面信息时, 访问那个目标进程的页面信息*

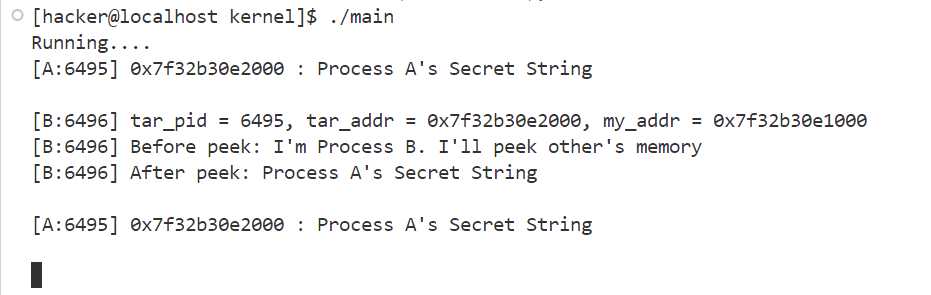
*然后每一步都做了失败处理, 这个没什么好说的*

SYSCALL\_DEFINE3(peep\_page, pid\_t, tar\_pid\_nr, unsigned long, tar\_addr, unsigned long, my\_addr)  
{  
 long rc = 0;  
  
 // 开始  
 printk(KERN\_INFO "peep\_page: called with tar\_pid\_nr=%d, tar\_addr=%lx, my\_addr=%lx\n", tar\_pid\_nr, tar\_addr, my\_addr);  
  
 // 获取目标进程的 task\_struct 结构  
 struct task\_struct \*tar\_task = find\_get\_task\_by\_vpid(tar\_pid\_nr);  
 // 如果失败  
 if (!tar\_task) {  
 rc = -ESRCH;  
 printk(KERN\_WARNING "peep\_page: target task with pid %d not found\n", tar\_pid\_nr);  
 return rc;  
 }  
  
 // 获取目标进程的内存描述符 mm\_struct  
 struct mm\_struct \*tar\_mm = get\_task\_mm(tar\_task);  
 // 失败处理  
 if (!tar\_mm) {  
 rc = -EACCES;  
 printk(KERN\_WARNING "peep\_page: failed to get mm\_struct for pid %d\n", tar\_pid\_nr);  
 goto FAIL\_TAR\_TASK;  
 }  
  
 // 锁定目标进程的内存映射，进行只读操作  
 mmap\_read\_lock(tar\_mm);  
  
 // 查找并扩展目标地址所在的虚拟内存区域  
 struct vm\_area\_struct \*vma = find\_extend\_vma(tar\_mm, tar\_addr);  
 // 失败处理  
 if (!vma) {  
 rc = -EFAULT;  
 printk(KERN\_WARNING "peep\_page: address %lx not found in VMA of pid %d\n", tar\_addr, tar\_pid\_nr);  
 goto FAIL\_TAR\_MM\_READ\_LOCK;  
 }  
  
 // 将目标地址转换为页表项  
 pte\_t \*ptep = addr\_to\_pte(tar\_mm, tar\_addr);  
 // 失败处理  
 if (!ptep) {  
 rc = -EFAULT;  
 printk(KERN\_WARNING "peep\_page: failed to get PTE for address %lx in pid %d\n", tar\_addr, tar\_pid\_nr);  
 goto FAIL\_TAR\_MM\_READ\_LOCK;  
 }  
  
 // 获取页表项的内容  
 pte\_t pte = \*ptep;  
 // 如果页表项不存在  
 if (!pte\_present(pte)) {  
 printk(KERN\_WARNING "peep\_page: page not present for address %lx in pid %d\n", tar\_addr, tar\_pid\_nr);  
 rc = -EBUSY;  
 goto FAIL\_TAR\_MM\_READ\_LOCK;  
 }  
  
 // 获取目标地址对应的物理页面  
 struct page \*page = vm\_normal\_page(vma, tar\_addr, pte);  
 // 如果物理页面不存在  
 if (!page) {  
 printk(KERN\_WARNING "peep\_page: normal page not found for address %lx in pid %d\n", tar\_addr, tar\_pid\_nr);  
 rc = -EEXIST;  
 goto FAIL\_TAR\_MM\_READ\_LOCK;  
 }  
  
 // 尝试获取页面的引用计数  
 rc = try\_grab\_page(page, FOLL\_GET);  
 // 失败处理  
 if (rc) {  
 printk(KERN\_WARNING "peep\_page: failed to grab page for address %lx in pid %d\n", tar\_addr, tar\_pid\_nr);  
 goto FAIL\_TAR\_MM\_READ\_LOCK;  
 }  
  
 // 将物理页面映射到内核地址空间  
 void \*kaddr = kmap\_local\_page(page);  
  
 // 将页面内容复制到用户空间地址  
 unsigned long n = copy\_to\_user((void \*) my\_addr, kaddr, PAGE\_SIZE);  
 // 失败处理  
 if (n) {  
 printk(KERN\_WARNING "peep\_page: failed to copy %ld bytes to user space address %lx\n", n, my\_addr);  
 rc = -EFAULT;  
 }  
  
 // 取消页面的内核地址映射  
 kunmap\_local(kaddr);  
  
 // 释放页面的引用计数  
 put\_page(page);  
  
 FAIL\_TAR\_MM\_READ\_LOCK:  
 // 解锁目标进程的内存映射  
 mmap\_read\_unlock(tar\_mm);  
  
 FAIL\_TAR\_MM:  
 // 释放目标进程的内存描述符  
 mmput(tar\_mm);  
  
 FAIL\_TAR\_TASK:  
 // 释放目标进程的 task\_struct 结构  
 put\_task\_struct(tar\_task);  
  
 // 返回结果码  
 return rc;  
}

*ii. 实验环境中测试程序的语义。*

#include <stdio.h>  
#include <unistd.h>  
#include <sys/mman.h>  
#include <string.h>  
  
// 子进程函数  
int sub(pid\_t tar\_pid, void \*tar\_addr)  
{  
 // 获取子进程的 PID  
 pid\_t my\_pid = getpid();  
  
 // 要写入内存的字符串  
 char \*str = "I'm Process B. I'll peek other's memory";  
  
 // 分配一段内存  
 char \*my\_addr = mmap(NULL, 4096, PROT\_READ|PROT\_WRITE, MAP\_PRIVATE|MAP\_ANON, -1, 0);  
 strcpy(my\_addr, str);  
  
 // 打印目标进程 PID、目标地址和子进程分配的内存地址  
 printf("[B:%d] tar\_pid = %d, tar\_addr = %p, my\_addr = %p\n", my\_pid, tar\_pid, tar\_addr, my\_addr);  
  
 // 打印读取内存前的内容  
 printf("[B:%d] Before peek: %s\n", my\_pid, my\_addr);  
  
 // 调用系统调用（假设编号为 548），尝试读取目标进程的内存  
 long ret = syscall(548, tar\_pid, tar\_addr, my\_addr);  
  
 // 如果系统调用成功，打印读取后的内容，否则打印错误信息  
 if (ret == 0) {  
 printf("[B:%d] After peek: %s\n\n", my\_pid, my\_addr);  
 } else {  
 printf("Syscall Wrong! ret = %ld\n\n", ret);  
 }  
  
 // 子进程进入死循环，每 5 秒打印一次  
 while (1) {  
 sleep(5);  
 }  
  
 return 0;  
}  
  
// 主函数  
int main(void)  
{  
 printf("Running....\n");  
  
 // 获取父进程的 PID  
 pid\_t pid = getpid();  
  
 // 要写入内存的字符串  
 char \*str = "Process A's Secret String";  
  
 // 分配一段内存  
 char \*addr = mmap(NULL, 4096, PROT\_READ | PROT\_WRITE, MAP\_PRIVATE | MAP\_ANON, -1, 0);  
 strcpy(addr, str);  
  
 // 创建子进程  
 pid\_t sub\_pid = fork();  
 if (sub\_pid == -1) {  
 // 如果 fork 失败，打印错误信息  
 printf("Fork Error!\n");  
 return 1;  
 } else if (sub\_pid != 0) {  
 // 父进程进入死循环，每 5 秒打印一次内存内容  
 while (1) {  
 printf("[A:%d] %p : %s\n\n", pid, addr, addr);  
 sleep(5);  
 }  
 } else {  
 // 子进程调用 sub 函数  
 sub(pid, addr);  
 }  
  
 return 0;  
}

*iii. 测试结果的说明*



成功窥探了父进程的存储 Secret String 的页面信息

# 4 个人总结

*i.此次实验活动中遇到的哪些关键问题，自己是如何发现并解决的。*

本次实验遇到了非常多非常多的问题, 以至于到去验收的时候都没有完全解决, 给我造成了极大困扰, 同时也学到了很多东西

首先基础题的第二题, 由于对于 信号量, 忙等待 的代码实现并不熟悉, 花了一些时间去学习

然后对于 bpftrace 的几题, 似乎中文互联网上信息较少, 花了很长时间去检索 bpftrace 脚本的语法, 有哪些探针, 分别有什么作用这些信息, 好在最后还是完成了问题

最后是设计内核编写编译安装的三题, 三题都对我造成了一些困扰

首先是 rr 调度, 我知道自己写的有问题, 但是由于编译 + 安装一次时间成本太高, 我又没有掌握其他验证调试的方法, 所以到最后也没有将其修改正确

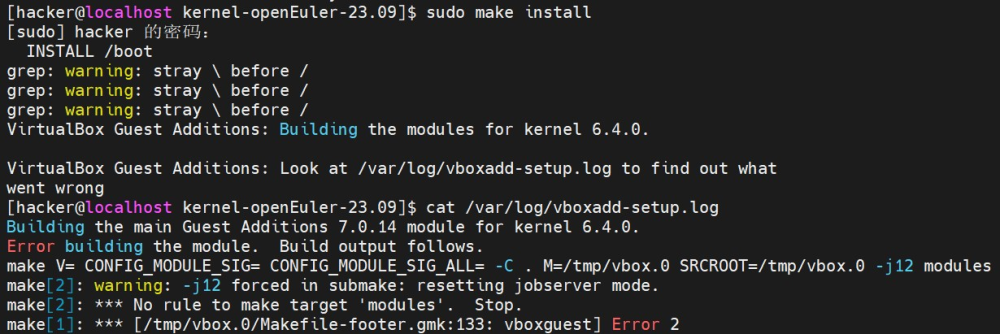
然后是 中级题 的最后一题的第一个, 起初没有思路, 好在看了高级题的第二题以后照猫画虎解决了

最后是 高级题 的第二题, 花了很长时间去学习各个结构体有哪些成员函数, 变量

对于虚拟机的使用同样带来了很多问题

首先是 vmware 给 open\_euler 分配的硬盘大小默认为 50GB, 在 sudo make all 后几乎占用 100%, 后续操作会出现因为硬盘没有空余空间报错, 花了一段时间去学习 linux 的给分区扩容的命令

最后的一个问题如下



这里的信息让我误以为内核没有成功安装, 花了很长时间尝试去解决这个问题, 最后发现不需要解决

*ii.上述问题的发现与解决对自己有哪些启发（从技术层面和学习做事方法两个角度进行说明）。*

技术层面:

我的技术还不够精湛, 对于 linux 操作系统, 很多方面还不够了解, 懂一点但不多, 还需要很多学习

学习做事方法:

感觉要扩展获取信息的渠道, 以能更好地应对实验过程中出现的各种出乎预料的报错信息