实验一：创建进程

（一）实验环境：Ubuntu 23.10，Linux kernel 6.5.0-28-generic

（二）实验内容：创建两个进程，让子进程读取一个文件，父进程等待子进程读取完文件后继续执行，实现进程协同工作。进程协同工作就是协调好两个进程，使之安排好先后次序并以此执行，可以用等待函数来实现这一点。当需要等待子进程运行结束时，可在父进程中调用等待函数。

（三）主要实现代码：

头文件

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>


int main(){
```

主体代码

```
int main(){
    int proc_id = 0;
    while((proc_id=fork()) == -1);

    if(proc_id>0){
        // 父
        printf("start waiting\n");
        proc_id = waitpid(proc_id, NULL, 0); // 等待子进程结束
        printf("parent_proc wait over\n");
        // prc
    } else {
        for(int i = 0; i < 100; i++){
            FILE* f = fopen("./content.txt", "r");
       }
        printf("child_proc read over\n");
    }
    return 0;
}
```

创建子进程后，根据 proc_id 来区分父子进程，父进程中使用 waitpid()函数来等待子进程执行结束，子进程反复读取 content.txt 文件 100 次。

（四）实验结果

```
(base) rongrong@rongrongUbuntu:~/os_exp/one$ gcc one.c
(base) rongrong@rongrongUbuntu:~/os_exp/one$ ls
a.out  content.txt  one.c
(base) rongrong@rongrongUbuntu:~/os_exp/one$ ./a.out
start waiting
child_proc read over
parent_proc wait over
(base) rongrong@rongrongUbuntu:~/os_exp/one$
```

实验二：线程共享进程数据

（一）实验环境：Ubuntu 23.10，Linux kernel 6.5.0-28-generic

（二）实验内容：在进程中定义全局共享数据，在线程中直接引用该数据进行更改并输出该数据；在进程中定义全局指针，利用该指针一个线程更改另一个线程的数据。

（三）主要实现代码：

头文件

```c
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
```

其中<pthread.h>用来创建线程

全局变量和指针

```c
int globalData = 0;
int* globalPointer = NULL;
```

两个线程各自的函数

```c
void* thread1(void* arg){
    // 用于修改进程数据
    printf("before fix: %d\n", globalData);
    globalData = 100;
    printf("after fix: %d\n", globalData);

    // 用于修改线程数据
    globalPointer = (int*)malloc(sizeof(int));
    *globalPointer = 10;
    return NULL;
}

void* thread2(void* arg){
    printf("before: %d\n", *globalPointer);
    *globalPointer = 204;
    printf("after: %d\n", *globalPointer);
    return NULL;
}
```

其中 thread1()用来修改进程的数据：修改为 100。用 malloc()定义该线程的变量，并用全局指针指向该块空间，初值设为 10。

thread2()用来修改第一个线 malloc()申请出来的空间的数据，修改为 204。

主体 main 函数

```c
int main(){
    pthread_t threadId1, threadId2;
    int err = pthread_create(&threadId1, NULL, thread1, NULL);

    //int pthread_create(pthread_t *tid, const pthread_arrt_t* attr,
    //void*(*start_routine)(void *), void* arg);

    //int pthread_join(pthread_t thread, void **retval);//等待线程结束

    pthread_join(threadId1, NULL);
    err = pthread_create(&threadId2, NULL, thread2, NULL);
    pthread_join(threadId2, NULL);
    return 0;
}
```

pthread_create()函数用来创建线程，pthread_join 用来等待指定的线程结束，函数的参数见图。

分别创建两个线程后，再各自执行。

（四）实验结果

编译时添加参数-pthread，表示需要链接到’pthread’库以确保能够正确运行。

```
(base) rongrong@rongrongUbuntu:~/os_exp/two$ gcc two.c -o two.out -pthread
(base) rongrong@rongrongUbuntu:~/os_exp/two$ ls
two.c  two.out
(base) rongrong@rongrongUbuntu:~/os_exp/two$ ./two.out
before fix: 0
after fix: 100
before: 10
after: 204
```

实验三：信号通信

（一）实验环境：Ubuntu 23.10，Linux kernel 6.5.0-28-generic

（二）实验内容：父进程创建一个有名事件，由子进程发送事件信号，父进程获取事件信号后进行相应的处理。

（三）主要实现代码：

头文件

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <signal.h>
```

其中<signal.h>用来进行信号通信。

收到信号后执行的代码

```
void sigchld_handler(int sig) {
    printf("this is the named event created by parent proc.\n");
    printf("recieving the signal.\n");
}

void bro_sigchld_handler(int sig) {
    printf("this is signal passing in bro to bro.\n");
}
```

其中 sigchld_handler()是子进程向父进程发送信号父进程执行的函数。

bro_sigchld_handler()是两个兄弟进程即父进程的两个子进程之间发送信号其中一个子进程执行的函数。

main 函数主体

```
int main() {
    int child_id1 = 0;
    int child_id2 = 0;
    // map信号和事件
    signal(SIGUSR2, sigchld_handler);
    // 创建子进程1
    while((child_id1 = fork()) == -1);
    if(child_id1 == 0 && child_id2 == 0) {
        // 第一个子进程序
        signal(SIGUSR1, bro_sigchld_handler);
    }

    if(child_id1 > 0) {
        // 创建子进程的兄弟进程
        while((child_id2 = fork()) == -1);
    }
    // parent_proc: child_id1 = pid1 child_id2 = pid2
    // child_proc1: child_id1 = 0 child_id2 = 0
    // child_proc2: child_id1 = pid1 child_id2 = 0

    if(child_id1 > 0 && child_id2 > 0) {
        // 父进程等待所有子进程执行
        waitpid(child_id1, NULL, 0);
        waitpid(child_id2, NULL, 0);
    }
```

首先 child_id1/2 用来接收子进程的 pid。signal()函数定义父进程接受到 SIGUSR2 信号后要执行的函数为 sigchld_handler()。

然后创建子进程 1，再对子进程 1 通过 signal()函数定义子进程 1 接收到 SIGUSR1 信号后执行的函数为 bro_sigchld_handler()。

接着创建子进程 2。此时在父进程，子进程 1，子进程 2 中的 child_id1 和 child_id2 如图中所示。在父进程中调用 waitpid()函数来等待两个子进程执行完毕。

```c
if(child_id1 > 0 && child_id2 == 0){
    // 第二个子进程
    printf("this is child proc2\n");
    // 向第一个子进程发送信号
    kill(child_id1, SIGUSR1);
    sleep(1);
    exit(1);
} else if(child_id1 == 0 && child_id2 == 0) {
    // 保证第二个子进程先执行
    sleep(2);
    // 第一个子进程
    printf("this is child proc1\n");
    // 向父进程发送信号
    kill(getppid(), SIGUSR2);
    sleep(1);
    exit(1);
} else if(child_id1 > 0 && child_id2 > 0) {
    printf("this is parent proc\n");
    exit(1);
}
return 0;
```

接下来子进程 2 通过 kill()函数向子进程 1 发送 SIGUSR1 信号，子进程 1 做出相应操作，对应兄弟进程间的通信。然后子进程 1 通过 kill()函数向父进程发送 SIGUSR2 信号，父进程执行相应操作。

（四）实验结果

```
(base) rongrong@rongrongUbuntu:~/os_exp/three$ gcc test.c
(base) rongrong@rongrongUbuntu:~/os_exp/three$ ./a.out
this is child proc2
this is signal passing in bro to bro.
this is child proc1
this is the named event created by parent proc.
recieving the signal.
this is parent proc
(base) rongrong@rongrongUbuntu:~/os_exp/three$
```

实验四：匿名管道通信

（一）实验环境：Ubuntu 23.10，Linux kernel 6.5.0-28-generic

（二）实验内容：分别建立名为 Parent 的单文档应用程序和 Child 的单文档应用程序作为父子进程，由父进程创建一个匿名管道，实现父子进程向匿名管道写入和读取数据。

（三）主要实现代码

头文件

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
```

其中<unistd.h>中定义了管道通信的函数接口。

```
int p_to_c[2];
```

定义全局变量，用于创建管道。

main 函数主体

```
// 创建匿名管道
pipe(p_to_c);

int proc_id = 0;
// 创建子进程
while((proc_id=fork()) == -1);
```

pipe()函数创建匿名管道，父进程再创建一个子进程。

匿名管道操作

```
if(proc_id == 0) {
    // 子进程
    waitpid(getppid(), NULL, 0);

    char content[50];
    // close write
    close(p_to_c[1]);
    read(p_to_c[0], content, sizeof(content));
    printf("child gets from parent %s\n", content);
    close(p_to_c[0]);
    exit(1);
}else if(proc_id > 0) {
    // 父进程
    char msg[] = "Hello, this is the specific content.";
    // 关闭读端
    close(p_to_c[0]);
    // 从写端写入
    write(p_to_c[1], msg, sizeof(msg));
    printf("parent's content: %s\n",msg);
    // 关闭写端
    close(p_to_c[1]);
}
return 0;
```

在父进程中，定义字符串 msg，close()函数关闭读端，write()函数从匿名管道写端写入 msg，再关闭写端。

在子进程中，定义 content，来接收从管道中获取的数据。close()关闭写端，read()从匿名管道读取数据，再关闭读端。

（四）实验结果

```
(base) rongrong@rongrongUbuntu:~/os_exp/four$ gcc parent.c
(base) rongrong@rongrongUbuntu:~/os_exp/four$ ./a.out
parent's content: Hello, this is the specific content.
child gets from parent Hello, this is the specific content.
```

实验五：命名匿名管道通信

（一）实验环境：Ubuntu 23.10，Linux kernel 6.5.0-28-generic

（二）实验内容：建立父子进程，由父进程创建一个命名匿名管道，由子进程向命名管道写入数据，由父进程从命名管道读取数据。

（三）主要实现代码：

头文件

```
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <string.h>
#include <fcntl.h>
#include <limits.h>
```

其中<sys/stat.h>中定义了创建命名管道的函数接口

main 函数主体

```
int pipe_id_p = 0;
int pipe_id_c = 0;
const char* fifo_name = "/home/rongrong/os_exp/five/fifo.tmp";

int res = 0;
// 创建命名管道
res = mkfifo(fifo_name, S_IFIFO);

//创建子进程
int proc_id = 0;
while((proc_id = fork()) == -1);

// 打开命名管道有4种方式
// open(const char *path, O_RDONLY); // 1
//open(const char *path, O_RDONLY | O_NONBLOCK); // 2
//open(const char *path, O_WRONLY); // 3
//open(const char *path, O_WRONLY | O_NONBLOCK); // 4
```

pipe_id_p/c 分别用来接收父子进程调用 open 打开命名管道的返回值，mkfifo()函数用来创建命名管道，fifo_name 为命名管道路径，S_IFIFO 表明创建的文件类型是 FIFO，即命名管道的类型。

open()函数打开命名管道的四种方式如图。

```c
if(proc_id == 0){
    // 子进程
    pipe_id_c = open(fifo_name, O_WRONLY);
    char st[] = "this is the content in the named pipe.";
    printf("child_p write to the named pipe: %s\n", st);
    if(pipe_id_c != -1){
        write(pipe_id_c, st, sizeof(st));
    } else {
        perror("open");
    }

    close(pipe_id_c);
} else if(proc_id > 0){
    // 父进程

    pipe_id_p = open(fifo_name, O_RDONLY);
    char msg[50] = {0};

    if(pipe_id_p != -1){
        read(pipe_id_p, msg, sizeof(msg));
    }
    close(pipe_id_p);
    printf("parent_p read from the named pipe: %s\n", msg);
}

return 0;
```

子进程以 O_WRONLY(阻塞只写)方式打开命名管道，并写入字符串 st。

父进程以 O_RDONLY(阻塞只读)方式打开命名管道，并读出内容写入 msg。

（四）实验结果

```
(base) rongrong@rongrongUbuntu:~/os_exp/five$ gcc five.c
(base) rongrong@rongrongUbuntu:~/os_exp/five$ sudo ./a.out
child_p write to the named pipe: this is the content in the named pipe.
parent_p read from the named pipe: this is the content in the named pipe.
(base) rongrong@rongrongUbuntu:~/os_exp/five$ ls
a.out  fifo.tmp  five.c  warning
(base) rongrong@rongrongUbuntu:~/os_exp/five$
```

其中用 sudo 是因为命名管道文件 fifo.tmp 操作权限限制。

实验六：信号量实现进程同步

（一）实验环境：Ubuntu 23.10，Linux kernel 6.5.0-28-generic

（二）实验内容：

- 生产者进程生产产品，消费者进程消费产品。
- 当生产者进程生产产品时，如果没有空缓冲区可用，那么生产者进程必须等待消费者进程释放出一个缓冲区。
- 当消费者进程消费产品时，如果缓冲区中没有产品，那么消费者进程将被阻塞，直到新的产品被生产出来。

（三）主要实现代码：

头文件

```c
#include <stdio.h>
#include <unistd.h>
#include <semaphore.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <sys/wait.h>
```

其中信号量函数的接口定义在<semaphore.h>中。

main 函数主体

```c
int main() {
    // sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);
    // O_CREAT|O_RDWR 没有就创建，有就打开
    // 0666访问权限
    // 5,0初始值
    sem_t* p = sem_open("/S1", O_CREAT|O_RDWR, 0666, 5);
    sem_t* s = sem_open("/S2", O_CREAT|O_RDWR, 0666, 0);
    sem_t* mutex = sem_open("/mutex",  O_CREAT|O_RDWR, 0666, 1);
    if(p == SEM_FAILED || s == SEM_FAILED) {
        perror("sem_open");
        exit(1);
    }
    // 信号量初值
    sem_init(p, 1, 5);
    sem_init(s, 1, 0);
    sem_init(mutex, 1, 1);

    int proc_id = 0;
    // 创建三个子进程作为消费者
    while((proc_id = fork()) == -1);
    while((proc_id = fork()) == -1);
    while((proc_id = fork()) == -1);

    int* current = NULL;
    int* current_p = NULL;
```

先用 sem_open()函数创建/S1,/S2,/mutex 三个信号量，/S1,/S2 分别为生产者消费者的私有信号量，/mutex 为共有信号量。

再用 sem_init()函数初始化每个信号量初值，/S1 为 5，/S2 为 0，/mutex 为 1，再创建三个子进程。这样之后，生产者有 5 个，消费者有 3 个。

```
if(proc_id == 0) {
    // 消费者
    while(1){
        sleep(1);
        sem_wait(s);//P
        sem_wait(mutex);//P
        printf("拿出一件产品\n");
        sem_post(mutex);//V
        sem_post(p);//V
        printf("消费产品\n");
        printf("out consumer, into producer\n\n");

    }
} else if(proc_id > 0){
    // 生产者
    while(1){
        sem_wait(p);//P
        sem_wait(mutex);//P
        printf("放入一件产品\n");
        sem_post(mutex);//V
        sem_post(s);//V
        printf("out producer, into consumer\n\n");
        sleep(0.5);
    }
}


return 0;
```

子进程为消费者，sem_wait()为 P 操作，sem_post()为 V 操作，先对/S2 进行 P，对/mutex 进行 P，处理完后再对/mutex 进行 V，对/S1 进行 V。

父进程为生产者，先对/S1 进行 P，对/mutex 进行 P，处理完后对/mutex 进行 V，对/S2 进行 V。

（四）实验结果：

```
(base) rongrong@rongrongUbuntu:~/os_exp/six$ gcc
a.out  sem.c
(base) rongrong@rongrongUbuntu:~/os_exp/six$ gcc sem.c
(base) rongrong@rongrongUbuntu:~/os_exp/six$ ./a.out
放入一件产品
out producer, into consumer

放入一件产品
out producer, into consumer

放入一件产品
out producer, into consumer

放入一件产品
out producer, into consumer

放入一件产品
out producer, into consumer

拿出一件产品
消费产品
放入一件产品
out producer, into consumer

out consumer, into producer

拿出一件产品
放入一件产品
out producer, into consumer

拿出一件产品
消费产品
out consumer, into producer

放入一件产品
out producer, into consumer

拿出一件产品
消费产品
```

实验七：共享主存实现进程通信

（一）实验环境：Ubuntu 23.10，Linux kernel 6.5.0-28-generic

（二）实验内容：为基于共享主存解决读者-写着问题，需要由写进程首先创建一个共享主存，并将该共享主存区映射到虚拟地址空间，随后读进程打开共享主存，并将该共享主存区映射到自己的虚拟地址空间，从中获取数据，并进行处理，以此实现进程通信。

（三）主要实现代码：

读者程序头文件

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/shm.h>
#include <sys/ipc.h>
```

其中<ipc.h>和<sys/shm.h>定义了用于获得标识符的函数接口以及操作共享内存的一些函数的接口。

读者 main 函数主体

```c
int main(){
    // key_t ftok(const char *pathname, int proj_id);
    // 根据路径和标识符proj_id生成key值用于共享内存
    key_t key = ftok(".", 1);

    if(key == -1) {
        perror("ftok");
        exit(1);
    }

    //int shmget(key_t  key, size_t  size, int shmflg)得到一个共享内存标识符或创建一个共享内存对
会报错

    //建立共享内存区
    int shmid;
    // 得到共享内存标识符
    shmid = shmget(key, 2*1024, IPC_CREAT|0666);
    if(shmid == -1){
        perror("shmget");
    }

    //void *shmat(int shmid, const void *shmaddr, int shmflg)连接共享内存标识符为shmid的共享内存
    //shmid共享内存标识符shmaddr指定共享内存出现在进程内存地址的什么位置，直接指定为NULL让内核自
    //Shmflg    规定主存的读写权限
    char s[20] = "wbb wudi";
    char* p = (char*)shmat(shmid, NULL, 0);
    for(int i = 0; i < strlen(s); i++){
        *(p+i) = s[i];
    }
    printf("writer writes: %s\n", s);
    //int shmdt(char*addr) 把共享主存从指定进程的虚拟地址空间断开。
    //addr是要断开连接的虚拟地址

    // 连接拆除
    int dt = shmdt(p);
    if(dt == -1){
        perror("shmdt");
    }
    return 0;

}
```

先用 ftok()函数获得标识符，再将该标识符传入 shmget()函数获得共享内存标识符 shmid，开了 2*1024 字节的空间，再调用 shmat()函数，传入共享内存标识符，将共享主存映射到虚拟地址空间，向共享主存写入字符串 s。最后调用 shmdt()函数拆除连接。

读者程序头文件

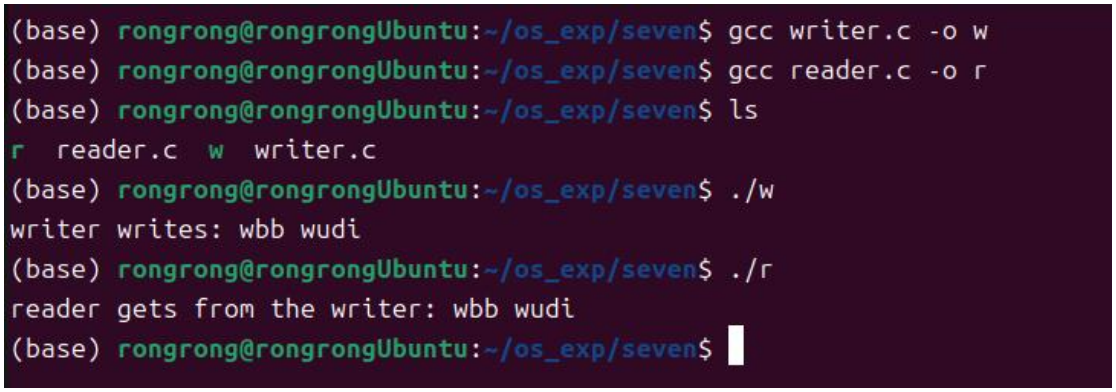
```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/shm.h>
#include <sys/ipc.h>
```

main 函数

```c
int main(){
    key_t key = ftok(".", 1);

    if(key == -1){
        perror("ftok");
        exit(1);
    }

    int shmid;
    shmid = shmget(key, 2*1024, IPC_CREAT|0666);
    if(shmid==-1){
        perror("shmget");
    }

    char* p = (char*)shmat(shmid, NULL, 0);
    char s[20] = {0};
    for(int i = 0; i < 8; i++){
        s[i] = *(p+i);
    }
    printf("reader gets from the writer: %s\n", s);
    return 0;
}
```

与写者操作流程一样，最后输出从共享内存中读出的数据。

（四）实验结果：



实验八：内核模式显示进程控制块信息

（一）实验环境：Ubuntu 23.10，Linux kernel 6.5.0-28-generic

（二）实验内容：在内核中，所有的进程控制块都被一个双向链表连接起来，该链表中的第一个进程控制块为 init_task。编写一个内核模块，模块接收用户传递的一个参数 num， num 指定要打印的进程控制块的 数量；若用户不指定 num 或者 num<0，模块则打印所有进程控制块的信息，需要打印的进程控制块信息由：进程 PID 和进程的可执行文件名

（三）主要实现代码：

```c
#include <linux/init.h>
#include <stdlib.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <asm/uaccess.h>
#include <linux/sched.h>
#include <linux/kernel.h>
#include<linux/sched/signal.h>
MODULE_LICENSE("GPL");

static int num = -1;

module_param(num, int, S_IRUGO);

static __init int exp_init(void){
  struct task_struct *p = NULL;
  if(num<0){
    for_each_process(p){
        printk("pid=%d,path=%s\n", p->pid, p->comm);
    }
  }else{
    for_each_process(p){
        num--;
        printk("pid=%d,path=%s\n", p->pid, p->comm);
        if(num==0){
            exit(1);
        }
    }

  }

  return 0;
}

static __exit void exp_exit(void){
  printk("Good bye.\n");
}

module_init(exp_init);
module_exit(exp_exit);
```

static int num = -1 为定义模块参数。如果不传入参数就默认输出全部进程。

MODULE_LICENSE("GPL");为模块许可声明，没有这句话的话加载模块会报 kernel tainted 的警告。

struct task_struct 为进程控制块的结构体，->指向 pid，comm 指向可执行文件名。

模块可通过 printk()函数将信息打印到系统日志/缓冲区中。

模块初始化和清理函数

module_init(exp_init); module_exit(exp_exit);

（四）实验结果

传入参数 num=5

```
(base) rongrong@rongrongUbuntu:~/os_exp/nine$ make
make -C /lib/modules/6.5.0-28-generic/build M=/home/rongrong/os_exp/nine modules
make[1]: 进入目录“/usr/src/linux-headers-6.5.0-28-generic”
warning: the compiler differs from the one used to build the kernel
  The kernel was built by: x86_64-linux-gnu-gcc-13 (Ubuntu 13.2.0-4ubuntu3) 13.2.0
  You are using:           gcc-13 (Ubuntu 13.2.0-4ubuntu3) 13.2.0
  CC [M]  /home/rongrong/os_exp/nine/listprocess.o
  MODPOST /home/rongrong/os_exp/nine/Module.symvers
  CC [M]  /home/rongrong/os_exp/nine/listprocess.mod.o
  LD [M]  /home/rongrong/os_exp/nine/listprocess.ko
  BTF [M] /home/rongrong/os_exp/nine/listprocess.ko
Skipping BTF generation for /home/rongrong/os_exp/nine/listprocess.ko due to unavailability of vmlinux
make[1]: 离开目录“/usr/src/linux-headers-6.5.0-28-generic”
(base) rongrong@rongrongUbuntu:~/os_exp/nine$ sudo insmod listprocess.ko num=5
[sudo] rongrong 的密码：
(base) rongrong@rongrongUbuntu:~/os_exp/nine$ sudo dmesg
[ 3538.279882] Good bye.
[ 3628.987100] pid=1,path=systemd
[ 3628.987104] pid=2,path=kthreadd
[ 3628.987105] pid=3,path=rcu_gp
[ 3628.987106] pid=4,path=rcu_par_gp
[ 3628.987107] pid=5,path=slub_flushwq
(base) rongrong@rongrongUbuntu:~/os_exp/nine$
```

不传参数

```
(base) rongrong@rongrongUbuntu:~/os_exp/nine$ sudo insmod listprocess.ko
(base) rongrong@rongrongUbuntu:~/os_exp/nine$ sudo dmesg -c
[ 3695.611780] pid=1,path=systemd
[ 3695.611784] pid=2,path=kthreadd
[ 3695.611785] pid=3,path=rcu_gp
[ 3695.611786] pid=4,path=rcu_par_gp
[ 3695.611787] pid=5,path=slub_flushwq
[ 3695.611788] pid=6,path=netns
[ 3695.611789] pid=11,path=mm_percpu_wq
[ 3695.611790] pid=12,path=rcu_tasks_kthre
[ 3695.611791] pid=13,path=rcu_tasks_rude_
[ 3695.611792] pid=14,path=rcu_tasks_trace
[ 3695.611793] pid=15,path=ksoftirqd/0
[ 3695.611794] pid=16,path=rcu_preempt
[ 3695.611794] pid=17,path=migration/0
[ 3695.611795] pid=18,path=idle_inject/0
[ 3695.611796] pid=19,path=cpuhp/0
[ 3695.611797] pid=20,path=cpuhp/1
[ 3695.611798] pid=21,path=idle_inject/1
[ 3695.611799] pid=22,path=migration/1
[ 3695.611800] pid=23,path=ksoftirqd/1
[ 3695.611801] pid=26,path=kdevtmpfs
[ 3695.611802] pid=27,path=inet_frag_wq
[ 3695.611803] pid=29,path=kauditd
[ 3695.611804] pid=31,path=khungtaskd
[ 3695.611805] pid=33,path=oom_reaper
[ 3695.611806] pid=34,path=writeback
[ 3695.611807] pid=35,path=kcompactd0
[ 3695.611807] pid=36,path=ksmd
[ 3695.611808] pid=37,path=khugepaged
[ 3695.611809] pid=38,path=kintegrityd
[ 3695.611810] pid=39,path=kblockd
[ 3695.611811] pid=40,path=blkcg_punt_bio
[ 3695.611812] pid=41,path=tpm_dev_wq
[ 3695.611813] pid=42,path=ata_sff
[ 3695.611814] pid=43,path=md
[ 3695.611815] pid=44,path=md_bitmap
[ 3695.611816] pid=45,path=edac-poller
[ 3695.611817] pid=46,path=devfreq_wq
[ 3695.611818] pid=47,path=watchdogd
[ 3695.611819] pid=50,path=kswapd0
[ 3695.611820] pid=51,path=ecryptfs-kthrea
[ 3695.611821] pid=52,path=kthrotld
[ 3695.611821] pid=53,path=irq/24-pciehp
```