乌有理工大学

《操作系统》课程实验报告

实验题目	**实验二：进程管理**		
姓　　名	**Nemo**	**学　　号**	**2019114514**
组　　别	**–**	**合 作 者**	**–**
班　　级	**计科 NaN 班**	**指导教师**	**Outis**

1 实验概述

1.1 实验介绍

本实验通过在内核态创建进程，读取系统 CPU 负载，打印系统当前运行进程 PID 以及读取当前会话中的所有进程的信息，让学生们了解并掌握操作系统中的进程管理。另外通过编写进程 IPC 的典型算法，使学生掌握进程的同步互斥的基本原理和应用。

1.2 任务描述

- 编写一个内核模块，创建一个内核线程，并在模块退出时杀死该线程。
- 编写一个内核模块，实现读取系统一分钟内的 CPU 负载。
- 编写一个内核模块，打印当前系统处于运行状态的进程的 PID 和名字。
- 编写一个 misc 设备驱动，实现读取当前会话中的所有进程的信息。
- 编写用户态应用程序，解决生产者-消费者问题、理发师问题、读者-写者问题。

1.3 实验目的

- 正确编写满足功能的源文件，正确编译。
- 正常加载、卸载内核模块，且内核模块功能满足任务所述。
- 了解操作系统的进程管理。
- 了解操作系统进程控制块的结构。
- 掌握操作系统中的进程同步和互斥。

2 实验内容

2.1 编写小型内核模块

用 `make` 命令编译 kthread.c、cpu_loadavg.c 和 process_info.c，用 `insmod` 命令加载编译完成的内核模块，用 `rmmod` 命令卸载内核模块，用 `dmesg` 命令查看内核模块在

内核态的运行结果。

实验结果如下所示（openEuler 20.03）：

```
[root@ostest ~]# cd lab2
[root@ostest lab2]# cd task1
[root@ostest task1]# make
make -C /root/kernel M=/root/lab2/task1 modules
make[1]: Entering directory '/root/kernel'
  Building modules, stage 2.
  MODPOST 1 modules
make[1]: Leaving directory '/root/kernel'
[root@ostest task1]# insmod kthread.ko
[root@ostest task1]# rmmod kthread.ko
[root@ostest task1]# dmesg | tail -n5
[   57.868041] New kthread is running.
[   59.884021] New kthread is running.
[   61.899996] New kthread is running.
[   63.915991] New kthread is running.
[   65.859750] Kill new kthread.
[root@ostest task1]# cd ../task2
[root@ostest task2]# make
make -C /root/kernel M=/root/lab2/task2 modules
make[1]: Entering directory '/root/kernel'
  Building modules, stage 2.
  MODPOST 1 modules
make[1]: Leaving directory '/root/kernel'
[root@ostest task2]# insmod cpu_loadavg.ko
[root@ostest task2]# rmmod cpu_loadavg.ko
[root@ostest task2]# dmesg | tail -n3
[  112.088785] Start cpu_loadavg!
[  112.089046] The cpu loadavg in one minute is: 0.74
[  129.420341] Exit cpu_loadavg!
[root@ostest task2]# cd ../task3
[root@ostest task3]# make
make -C /root/kernel M=/root/lab2/task3 modules
make[1]: Entering directory '/root/kernel'
  Building modules, stage 2.
  MODPOST 1 modules
make[1]: Leaving directory '/root/kernel'
[root@ostest task3]# insmod process_info.ko
[root@ostest task3]# rmmod process_info.ko
[root@ostest task3]# dmesg | tail -n4
[  198.878576] Start process_info!
[  198.878871] 1)name:kworker/1:3  2)pid:452  3)state:0
[  198.879218] 1)name:insmod  2)pid:3686  3)state:0
[  205.743527] Exit process_info!
```

2.2 编写 misc 设备驱动

2.2.1 编写模块

内核模块安装好后，会创建一个设备文件 /dev/proc_relate。测试程序 test 执行时，打开这个设备文件，读取当前会话中用运行的所有进程信息，并显示出来。

模块的核心部分如下所示。其中，`current` 是内核中扩展到 `get_current()` 的宏，返回当前进程 `task_struct` 的指针。`for_each_process(p)` 是内核中扩展到 `for (p = &init_task ; (p = next_task(p)) != &init_task ; )` 的宏，由此来遍历 `task_struct` 链表。

```c
static ssize_t proc_relate_read(struct file *file, char __user *out,
                                size_t size, loff_t *off) {
  struct proc_info *buf = file->private_data;
  int cur_sessionid = current->sessionid;
  struct task_struct *p;
  int cnt = 0;
  struct proc_info *info = buf;
  for_each_process(p) {
    if (p->sessionid == cur_sessionid) {
      info->state = 0;
      info->pid = p->pid;
      info->tgid = p->tgid;
      strncpy(info->comm, p->comm, 16);
      info->prio = p->prio;
      info->static_prio = p->static_prio;
      info->mm = p->mm;
      info->active_mm = p->active_mm;
      info->sessionid = p->sessionid;
      info->real_parent = p->real_parent->pid;
      info->parent = p->parent->pid;
      info->group_leader = p->group_leader->pid;
      info++;
      cnt++;
      if (cnt >= 30) {
        break;
      }
    }
  }
  copy_to_user(out, buf, sizeof(struct proc_info) * cnt);
  return sizeof(struct proc_info) * cnt;
}
```

2.2.2 运行测试

编写 run.sh，并执行命令 `chmod +x run.sh`。

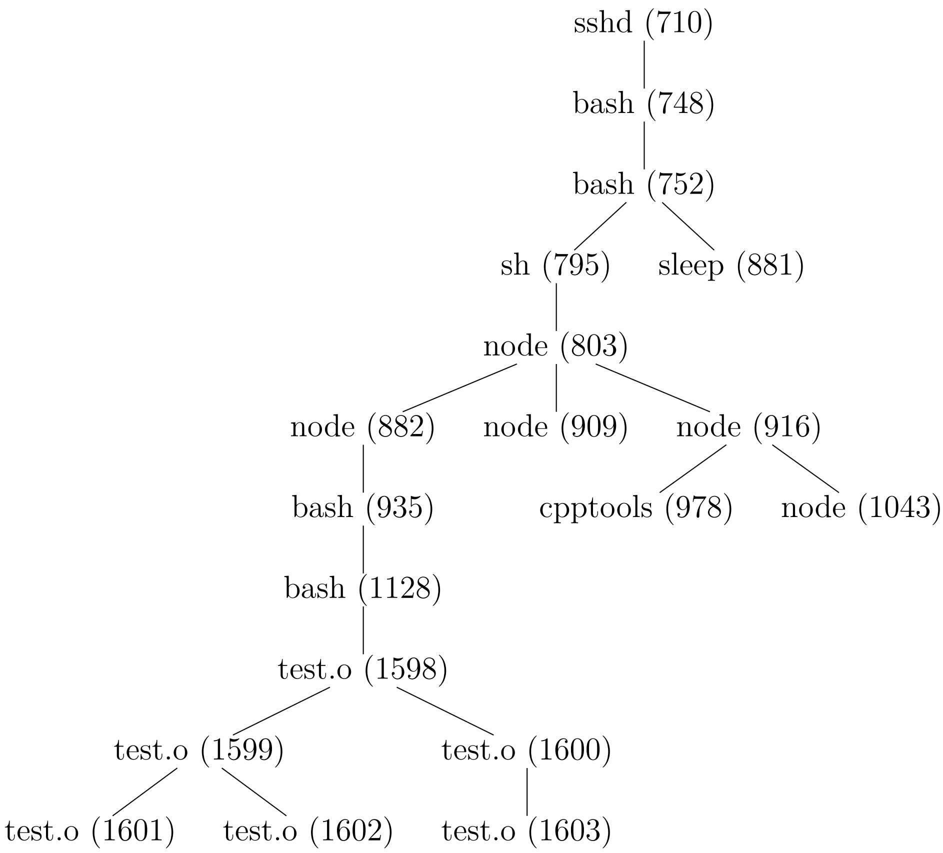
执行 run.sh，实验结果如下所示（openEuler 20.03）：

```
make
insmod proc_relate.ko
gcc test.c -std=c99 -o test.o -lm -lpthread
./test.o
rmmod proc_relate.ko
```

```
[root@ostest proc_relate]# ./run.sh
make -C /lib/modules/4.19.188/build M=/root/lab2/proc_relate modules
make[1]: Entering directory '/root/kernel'
  Building modules, stage 2.
  MODPOST 1 modules
make[1]: Leaving directory '/root/kernel'
here is parent process, pid = 1598
-------------------------------------------------------
pid = 710        tgid = 710       comm = sshd      sessionid = 1
mm = 0xffff8e53b39ad100 active_mm = 0xffff8e53b39ad100
parent = 535     real_parent = 535       group_leader = 710
-------------------------------------------------------
pid = 748        tgid = 748       comm = bash      sessionid = 1
mm = 0xffff8e53b6c74cc0 active_mm = 0xffff8e53b6c74cc0
parent = 710     real_parent = 710       group_leader = 748
...↓
```

```
...↑
-------------------------------------------------------
pid = 752       tgid = 752      comm = bash     sessionid = 1
mm = 0xffff8e53b6c75100 active_mm = 0xffff8e53b6c75100
parent = 748    real_parent = 748       group_leader = 752
-------------------------------------------------------
pid = 795       tgid = 795      comm = sh       sessionid = 1
mm = 0xffff8e53b6c75540 active_mm = 0xffff8e53b6c75540
parent = 752    real_parent = 752       group_leader = 795
-------------------------------------------------------
pid = 803       tgid = 803      comm = node     sessionid = 1
mm = 0xffff8e53b6c77b80 active_mm = 0xffff8e53b6c77b80
parent = 795    real_parent = 795       group_leader = 803
-------------------------------------------------------
pid = 881       tgid = 881      comm = sleep    sessionid = 1
mm = 0xffff8e53b6c75980 active_mm = 0xffff8e53b6c75980
parent = 752    real_parent = 752       group_leader = 881
-------------------------------------------------------
pid = 882       tgid = 882      comm = node     sessionid = 1
mm = 0xffff8e53b6c76200 active_mm = 0xffff8e53b6c76200
parent = 803    real_parent = 803       group_leader = 882
-------------------------------------------------------
pid = 909       tgid = 909      comm = node     sessionid = 1
mm = 0xffff8e53b6c74000 active_mm = 0xffff8e53b6c74000
parent = 803    real_parent = 803       group_leader = 909
-------------------------------------------------------
pid = 916       tgid = 916      comm = node     sessionid = 1
mm = 0xffff8e53b6c75dc0 active_mm = 0xffff8e53b6c75dc0
parent = 803    real_parent = 803       group_leader = 916
-------------------------------------------------------
pid = 935       tgid = 935      comm = bash     sessionid = 1
mm = 0xffff8e53b6c74880 active_mm = 0xffff8e53b6c74880
parent = 882    real_parent = 882       group_leader = 935
-------------------------------------------------------
pid = 978       tgid = 978      comm = cpptools sessionid = 1
mm = 0xffff8e53b3ae5540 active_mm = 0xffff8e53b3ae5540
parent = 916    real_parent = 916       group_leader = 978
-------------------------------------------------------
pid = 1043      tgid = 1043     comm = node     sessionid = 1
mm = 0xffff8e53b3ae7b80 active_mm = 0xffff8e53b3ae7b80
parent = 916    real_parent = 916       group_leader = 1043
-------------------------------------------------------
pid = 1100      tgid = 1100     comm = cpptools-srv     sessionid = 1
mm = 0xffff8e53b3ae7740 active_mm = 0xffff8e53b3ae7740
parent = 995    real_parent = 995       group_leader = 1100
-------------------------------------------------------
pid = 1128      tgid = 1128     comm = bash     sessionid = 1
mm = 0xffff8e53b3ae7300 active_mm = 0xffff8e53b3ae7300
parent = 935    real_parent = 935       group_leader = 1128
-------------------------------------------------------
pid = 1598      tgid = 1598     comm = test.o   sessionid = 1
mm = 0xffff8e53b3ae4cc0 active_mm = 0xffff8e53b3ae4cc0
parent = 1128   real_parent = 1128      group_leader = 1598
-------------------------------------------------------
pid = 1599      tgid = 1599     comm = test.o   sessionid = 1
mm = 0xffff8e53b3ae5dc0 active_mm = 0xffff8e53b3ae5dc0
parent = 1598   real_parent = 1598      group_leader = 1599
-------------------------------------------------------
[root@ostest proc_relate]# this is a child, pid is 1599
this is a child, pid is 1602
this is a child, pid is 1601
this is another child, pid is 1600
this is a child, pid is 1603
In thread, pid = 1600, tid = 1604, thread id = 139809071036160
thread has ended.
```

根据结果得到如下所示的进程树（sessionid 为 1）：



挑选编号为 803 和编号为 978 的进程，使用命令 `vi /proc/803/stat` 和 `vi /proc/978/stat`，结果如下。

在 https://man7.org/linux/man-pages/man5/proc.5.html 查阅 stat 文件各项的含义，发现该进程各信息与从 `task_struct` 中读取的一致。

```
803 (node) S 795 748 748 0 -1 4194304 39561 3653 289 2 260 43 5 1 20 0 12 0 3630 869892096
14677 18446744073709551615 4194304 69529246 140735566098384 0 0 0 0 16777216 87554 0 0 0 17
0 0 0 47 0 0 71628344 71710896 105033728 140735566105654  140735566105973
140735566105973 140735566106545 0
```

```
978 (cpptools) S 916 748 748 0 -1 1077936128 46078 41860 74 1083 362 144 97 54 20 0 19 0 4014
709951488 21059 18446744073709551615 4194304 18986461 140722810809024 0 0 0 0 4096 65536 0 0
0 17 0 0 0 4 0 0 21085848 21371864 42659840 140722810812938 140722810813008
140722810813008 140722810814386 0
```

2.3 编写用户态 IPC 算法

2.3.1 生产者-消费者问题（2 个生产者，3 个消费者）

按照如下算法编写程序，在用户态下运行，得到生产者-消费者问题的模拟结果。

Algorithm 1: 生产

Input: 生产者编号 id

while 未中断 **do**
- 休眠随机时间...
- P($empty$)
- P($mutex$)
- $cur\_item :=$ 随机项目
- $buffer[buffer\_index] := cur\_item$
- 做一些其他事...
- V($mutex$)
- V($full$)

end

Algorithm 2: 消费

Input: 消费者编号 id

while 未中断 **do**
- 休眠随机时间...
- P($full$)
- P($mutex$)
- $buffer\_index := buffer\_index - 1$
- $cur\_item := buffer[buffer\_index]$
- 做一些其他事...
- V($mutex$)
- V($empty$)

end

```
PRODUCER    CONSUMER      STOCK
 1 -> A                 A(1), a(0)
             2 <- A     A(0), a(0)
 1 -> a                 A(0), a(1)
 1 -> A                 A(1), a(1)
 1 -> A                 A(2), a(1)
 0 -> a                 A(2), a(2)
             4 <- a     A(2), a(1)
 1 -> A                 A(3), a(1)
             4 <- A     A(2), a(1)
 0 -> A                 A(3), a(1)
             3 <- A     A(2), a(1)
             2 <- A     A(1), a(1)
 1 -> a                 A(1), a(2)
 1 -> A                 A(2), a(2)
 1 -> a                 A(2), a(3)
             4 <- a     A(2), a(2)
 0 -> A                 A(3), a(2)
             3 <- A     A(2), a(2)
 1 -> a                 A(2), a(3)
...
```

2.3.2 理发师问题（5 个座位，10 个客人）

按照如下算法编写程序，在用户态下运行，得到理发师问题的模拟结果。

Algorithm 3: 理发

Input: 客人编号 id

$served\_customer\_cnt := 0$ **while** 未中断 **do**
- P($customer\_ready$) // 等待一个客人
- P($seat\_ready$) // 等待一个座位
- $avaliable\_seats\_cnt := avaliable\_seats\_cnt + 1$ // 占领一个座位
- V($seat\_ready$) // 座位已就绪
- P($mutex$) // 理发开始，上锁
- 做一些其他事...
- V($mutex$) // 理发结束，解锁
- V($barber\_ready$) // 理发师已就绪
- $served\_customer\_cnt := served\_customer\_cnt + 1$
- **if** $served\_customer\_cnt + unlucky\_customer\_cnt = NUM\_CUSTOMER$ **then**
  - 中断
- **end**

end

Algorithm 4: 等候

Input: 客人编号 id

P($seat\_ready$) // 等待一个座位

if $avaliable\_seats\_cnt > 0$ **then**
- $avaliable\_seats\_cnt := avaliable\_seats\_cnt - 1$ // 客人占领座位
- 做一些其他事...
- V($customer\_ready$) // 客人已就绪
- V($seat\_ready$) // 客人获得座位
- P($barber\_ready$) // 客人等待理发师

end

else // 没有空座位
- V($seat\_ready$) // 不用再等待座位了
- $unlucky\_customer\_cnt := unlucky\_customer\_cnt + 1$ // 客人离开

end

```
Customer 0 is waiting...  Avl = 4
Customer 1 is waiting...  Avl = 4
Customer 2 is waiting...  Avl = 3
Customer 3 is waiting...  Avl = 2
Customer 0 is OK.         Avl = 3
Customer 4 is waiting...  Avl = 2
Customer 5 is waiting...  Avl = 1
Customer 6 is waiting...  Avl = 0
Customer 1 is OK.         Avl = 1
Customer 7 is waiting...  Avl = 0
Customer 8 left.
Customer 9 left.
Customer 2 is OK.         Avl = 0
Customer 3 is OK.         Avl = 2
...↓
```

```
...↑
Customer 4 is OK.         Avl = 3
Customer 5 is OK.         Avl = 4
Customer 6 is OK.         Avl = 5
Customer 7 is OK.         Avl = 5
```

2.3.3 读者-写者问题（5 个读者，10 个写者）

按照如下算法编写 C 程序，在用户态下运行程序，得到读者-写者问题的模拟结果。

Algorithm 5: 读

```
Input: 读者编号 id
while 未中断 do
    P(read_try) // 读者等候
    P(read_mutex) // 锁读取
    read_cnt := read_cnt + 1
    if 该读者是第一个 then
        P(resource) // 锁资源
    end
    V(read_mutex) // 解锁读取
    V(reader) // 读者读完成
    做一些其他事...
    P(read_mutex) // 锁读取
    read_cnt := read_cnt − 1
    if 该读者是最后一个 then
        P(resource) // 解锁资源
    end
    V(read_mutex) // 解锁读取
end
```

Algorithm 6: 写

```
Input: 写者编号 id
while 未中断 do
    P(write_mutex) // 锁写入
    write_cnt := write_cnt + 1
    if 该写者是第一个 then
        P(read_try) // 读者不能尝试读取
    end
    P(resource) // 锁资源
    做一些其他事...
    V(resource) // 解锁资源
    P(write_mutex) // 解锁写入
    write_cnt := write_cnt − 1
    if 该写者是最后一个 then
        V(read_try) // 读者可以尝试读取
    end
    V(write_mutex) // 解锁读取
end
```

```
0 -> Resource (WRITE)
4 -> Resource (WRITE)
1 -> Resource (WRITE)
3 -> Resource (WRITE)
2 -> Resource (WRITE)
8 <- Resource (READ)
5 <- Resource (READ)
9 <- Resource (READ)
6 <- Resource (READ)
12 <- Resource (READ)
13 <- Resource (READ)
14 <- Resource (READ)
7 <- Resource (READ)
10 <- Resource (READ)
11 <- Resource (READ)
1 -> Resource (WRITE)
11 <- Resource (READ)
3 -> Resource (WRITE)
6 <- Resource (READ)
12 <- Resource (READ)
9 <- Resource (READ)
11 <- Resource (READ)
11 <- Resource (READ)
4 -> Resource (WRITE)
13 <- Resource (READ)
10 <- Resource (READ)
4 -> Resource (WRITE)
7 <- Resource (READ)
5 <- Resource (READ)
2 -> Resource (WRITE)
12 <- Resource (READ)
...
```

3 小结

通过在内核态创建进程，了解并掌握了操作系统中的进程管理。通过编写进程 IPC 的典型算法，掌握了进程的同步互斥的基本原理和应用。

本实验在华为弹性云服务器 root@121.36.*.*（openEuler 20.03）完成。