# 广州大学学生实验报告

**开课学院及实验室：**计算机科学与教育软件学院计算机软件实验室 2020**年**3**月**12**日**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **学院** | 计算机科学与教育软件学院 | **年级/专业/班** | 软件171 | **姓名** | 谢金宏 | **学号** | 1706300001 |
| **实验课**  **程名称** | Linux系统实验 | | | | | **成绩** |  |
| **实验项**  **目名称** | 进程控制与进程互斥 | | | | | **指导**  **老师** | 张艳玲 |

## 实验目的

1. 理解Linux下进程的结构；
2. 理解Linux下产生新进程的方法（系统调用——fork函数）；
3. 掌握如何启动另一程序的执行；
4. 理解Linux下线程的结构；
5. 理解Linux下产生新线程的方法；
6. 理解Linux系统下多进程与多线程的区别。

## 实验内容

1. 利用fork函数创建新进程，并根据fork函数的返回值，判断自己是处于父进程还是子进程中；
2. 在新创建的子进程中，使用exec类的函数启动另一程序的执行；分析多进程时系统的运行状态和输出结果；
3. 利用函数pthread\_create、pthread\_join和pthread\_exit编写一个最简单的多线程程序。理解多线程的运行和输出情况；
4. 利用信号量机制控制多线程的运行顺序，并实现多线程中数据的共享；
5. 分析Linux系统下多进程与多线程中的区别；
6. 用fork写一个测试程序，从父进程和子进程中打印出各自的PID，并把wait(), exit()系统调用加进去，使子进程返回退出状态给父进程，并将它包含在父进程的打印信息中。对运行结果进行分析；
7. 用代码编辑器查看自己计算机上的内核源代码中的include/linux/sched.h 文件中的task\_struct 结构，列出类型为list\_head 的字段，并说明哪几个字段可以形成进程的树结构，通过pstree 命令查看自己机器上的进程树结构；
8. 在Linux中用nproc或者lscpu 命令来查看CPU 核心数，多个核心数就意味着系统可以在同一时间处理多个进程；
9. 在Linux中用top或者uptime 命令来查看系统负载情况，返回的load average的3个数字表示的系统在1分钟、5分钟、15分钟内平均负载情况。

## 实验涉及的系统调用以及内核函数

1. **fork函数** 创建一个新进程。

系统调用格式：*pid=fork()*

参数定义：*int fork()*

fork()返回值意义如下：

0：在子进程中，pid变量保存的fork()返回值为0，表示当前进程是子进程。

>0：在父进程中，pid变量保存的fork()返回值为子进程的id值（进程唯一标识符）。

-1：创建失败。

1. **exec类的函数** 启动另一程序的执行。

Exec函数族其中一个函数的原型为：

*int execl(const char \*path, const char \*args)*

用于执行参数path字符串所代表的文件路径，接下来的参数代表执行该文件时传递过去的argv[0]、argv[1]……，最后一个参数必须用NULL结束。

如果执行成功则函数不会返回，执行失败则返回-1；失败的原因存于error中。

1. **pthread\_create函数** 创建一个线程。

函数原型为：

*extern int pthread\_create \_\_P ((pthread\_t \*\_\_thread, \_\_const pthread\_attr\_t \*\_\_attr,void \*(\*\_\_start\_routine) (void \*), void \*\_\_arg));*

第一个参数为指向线程标识符的指针，第二个参数用来设置线程属性，第三个参数是线程运行函数的起始地址，最后一个参数是运行函数的参数。

1. **pthread\_join函数** 等待一个线程的结束。

函数原型为：

*extern int pthread\_join \_\_P ((pthread\_t \_\_th, void \*\*\_\_thread\_return));*

第一个参数为被等待的线程标识符，第二个参数为一个用户定义的指针，它可以用来存储被等待线程的返回值。

这个函数是一个线程阻塞的函数，调用它的函数将一直等待到被等待的线程结束为止，当函数返回时，被等待线程的资源被收回。一个线程的结束有两种途径，一种是函数结束了，调用它的线程也就结束了；另一种方式是通过函数pthread\_exit来实现。

1. **sem\_init函数** 用来初始化一个信号量。

它的原型为：

*extern int sem\_init \_\_P ((sem\_t \*\_\_sem, int \_\_pshared, unsigned int \_\_value));*

sem为指向信号量结构的一个指针；pshared不为０时此信号量在进程间共享，否则只能为当前进程的所有线程共享；value给出了信号量的初始值。

## 实验原理与重点

1. **Linux系统下多进程编程：**
   1. 理解Linux下进程的结构：

Linux系统中一个进程在用户内存里有三部份的数据，就是“数据段”，“堆栈段”和“代码段”，一般的CPU, 如I386，都有上述三种段寄存器，以方便操作系统的运行。“代码段” 就是存放了程序代码的数据，堆栈段存放的就是子程序的返回地址、子程序的参数以及程序的局部变量。而数据段则存放程序的全局变量，常数以及动态数据分配的数据空间（比如用malloc之类的函数取得的空间）。系统如果同时运行数个相同的程序，它们之间就不能使用同一个堆栈段和数据段。

* 1. 利用fork（）函数创建进程和利用exec类的函数来启动进程的执行

一个程序调用fork函数，系统就为一个新的进程准备了前述三个段，首先，系统让新的进程与旧的进程使用同一个代码段，因为它们的程序还是相同的，对于数据段和堆栈段，系统则复制一份给新的进程，这样，父进程的所有数据都可以留给子进程，但是，子进程一旦开始运行，虽然它继承了父进程的一切数据，但实际上数据却已经分开，相互之间不再有影响了，也就是说，它们之间不再共享任何数据了。

一个进程一旦调用exec类函数，它本身就“死亡”了，系统把代码段替换成新的程序的代码，废弃原有的数据段和堆栈段，并为新程序分配新的数据段与堆栈段，唯一留下的，就是进程号，也就是说，对系统而言，还是同一个进程，不过已经是另一个程序了。

这样就实现了多进程编程和运行了。

1. **Linux系统下多线程编程**
   1. **为什么使用多线程？**

使用多线程的理由之一是和进程相比，它是一种非常“节俭”的多任务操作方式。在Linux系统下，启动一个新的进程必须分配给它独立的地址空间，建立众多的数据表来维护它的代码段、堆栈段和数据段，这是一种“昂贵”的多任务工作方式。而运行于一个进程中的多个线程，它们彼此之间使用相同的地址空间，共享大部分数据，启动一个线程所花费的空间远远小于启动一个进程所花费的空间，而且，线程间彼此切换所需的时间也远远小于进程间切换所需要的时间。据统计，总的说来，一个进程的开销大约是一个线程开销的30倍左右。

使用多线程的理由之二是线程间方便的通信机制。对不同进程来说，它们具有独立的数据空间，要进行数据的传递只能通过通信的方式进行，这种方式不仅费时，而且很不方便。线程则不然，由于同一进程下的线程之间共享数据空间，所以一个线程的数据可以直接为其它线程所用，这不仅快捷，而且方便。

当然，数据的共享也带来其他一些问题，有的变量不能同时被两个线程所修改，有的子程序中声明为static的数据更有可能给多线程程序带来灾难性的打击，这些正是编写多线程程序时最需要注意的地方。

多线程程序作为一种多任务、并发的工作方式，还有以下的优点：

1) 提高应用程序响应。这对图形界面的程序尤其有意义，当一个操作耗时很长时，整个系统都会等待这个操作，此时程序不会响应键盘、鼠标、菜单的操作，而使用多线程技术，将耗时长的操作（time consuming）置于一个新的线程，可以避免这种尴尬的情况。

2) 使多CPU系统更加有效。操作系统会保证当线程数不大于CPU数目时，不同的线程运行于不同的CPU上。

3) 改善程序结构。一个既长又复杂的进程可以考虑分为多个线程，成为几个独立或半独立的运行部分，这样的程序会利于理解和修改。

* 1. **多线程编程**

Linux系统下的多线程遵循POSIX线程接口，称为pthread。编写Linux下的多线程程序，需要使用头文件pthread.h，连接时需要使用库libpthread.a。

利用函数pthread\_create来创建一个线程，利用函数pthread\_join来等待一个线程的结束。多线程编程实例见实验指导书。

1. **利用信号量机制控制多线程的运行顺序（重点）**

信号量本质上是一个非负的整数计数器，它被用来控制对公共资源的访问。当公共资源增加时，调用函数sem\_post（）增加信号量。只有当信号量值大于０时，才能使用公共资源，使用后，函数sem\_wait（）减少信号量。

使用信号量实例见实验指导书。在这个例子中，一共有4个线程，其中两个线程负责从文件读取数据到公共的缓冲区，另两个线程从缓冲区读取数据作不同的处理（加和乘运算）。

## 实验原始数据、结果与分析

1. **利用fork函数创建新进程，并根据fork函数的返回值，判断自己是处于父进程还是子进程中。用ps命令查看系统中进程的运行状况，并分析输出结果。**

*// task1\_fork.c*

#include <stdio.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

int main(void) {

    int pid = -1;

    pid = fork();

    switch (pid)

    {

    case -1:

        printf("Fail to fork.\n");

        break;

    case 0:

        printf("This is child process, whose pid is %d.\n", getpid());

        execl("/bin/ps", "/bin/ps", "-o", "pid,ppid,comm", NULL);

        printf("Fail to exec!\n");

        break;

    default:

        printf("This is parent process, whose pid is %d.\n", getpid());

        wait(NULL);

        break;

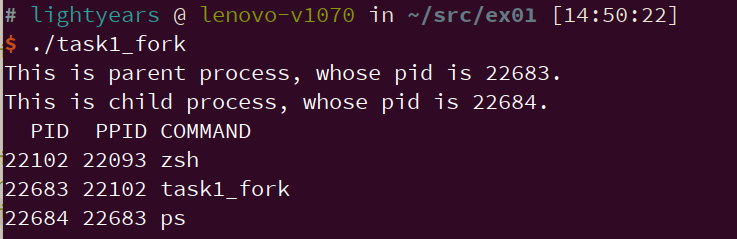
    }

}

在上面的代码中，主进程执行fork()创建了一个子进程。通过fork()的返回值可判断当前CPU处于主进程或子进程的上下文中。若fork的返回值为-1，则创建子进程失败；若fork()的返回值为0，则当前处于子进程的上下文中。若fork的返回值不是上面的两种情况，则fork返回的是子进程的PID，且当前处于主进程的上下文中。

在子进程中，先是打印子进程的ID，随后调用execl函数执行了ps命令。调用execl时，需要传递参数。第一个参数指明要调用的程序，随后的参数会传递给第一个参数指明的程序，最后以NULL结尾。另，根据POSIX的传统，传递给程序的第一参数应当为程序自身所在的路径。因此在上面的程序中，execl的第一个参数与第二个参数是相同的。子进程执行execl的效果相当于在shell中执行“/bin/ps -o pid,ppid,comm”，作用是显示当前运行的进程的PID（进程ID），PPID（父进程的PID）和COMM（命令）。若execl执行成功，则下面的“Fail to exec!”不会打印。

主进程中，先是打印主进程的ID，然后使用wait等待子进程结束。



1. **在新创建的子进程中，使用exec类的函数启动ps命令显示本机运行的所有进程的详细列表，并分析列表中不同进程的内存占用情况。**

*// task2\_exec.c*

#include <stdio.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

int main(void) {

    int pid = -1;

    pid = fork();

    switch (pid)

    {

    case -1:

        printf("Fail to fork.");

        break;

    case 0:

        execl("/bin/ps", "/bin/ps", "-e", "-o", "comm,pid,pcpu,pmem", "--sort", "-pmem", NULL);

        break;

    default:

        wait(NULL);

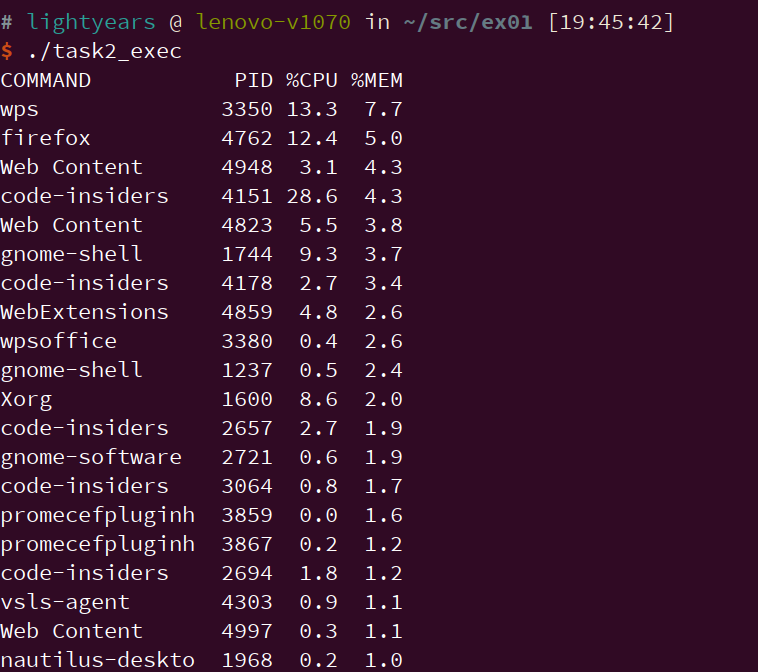
        break;

    }

}

本题代码与第1题类似，区别在于子进程执行execl时，向ps传递的参数不同。本体中所传参数相当于在shell中执行“/bin/ps -e -o comm,pid,pcpu,pmem --sort -pmem”。“-e”的作用是显示本机运行的所有线程，“-o comm,pid,pcpu,pmem”使输出结果包含命令、进程ID、CPU占用百分比和内存占用百分比。“--sort -pmem”使输出结果以内存占用百分比降序排列。

程序的部分输出如下图所示。可见当前系统中占用内存的前两个进程为wps和firefox。



1. **利用函数pthread\_create、pthread\_join和pthread\_exit编写一个最简单的多线程程序。理解多线程的运行和输出情况。**

*// task3\_threads.c*

#include <stdlib.h>

#include <stdio.h>

#include <pthread.h>

void thread(void)

{

    int i;

    for (i = 0; i < 3; i++) {

        printf("This is a pthread.\n");

    }

    srand(time(NULL));

    int \*retval = calloc(1, sizeof(int));

    \*retval = rand() % 256;

    printf("Passing return value: %d.\n", \*retval);

    pthread\_exit((void \*)retval);

}

int main(void)

{

    pthread\_t id;

    if (pthread\_create(&id, NULL, (void \*)thread, NULL) != 0)

    {

        printf("Fail to create pthreads!\n");

        exit(1);

    }

    for (int i = 0; i < 3; i++) {

        printf("This is the main thread.\n");

    }

    int \*retval = NULL;

    pthread\_join(id, (void \*\*)&retval);

    printf("The return value from pthread is %d.\n", \*retval);

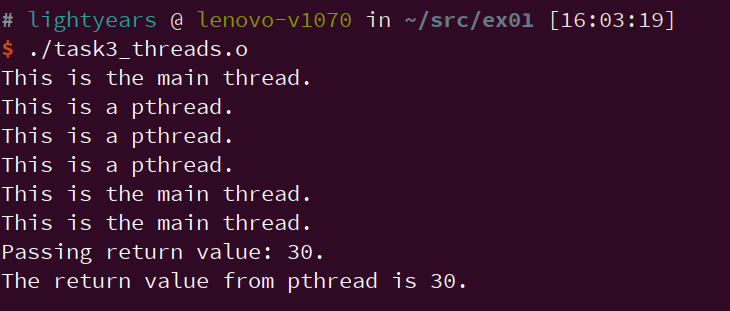
    free(retval);

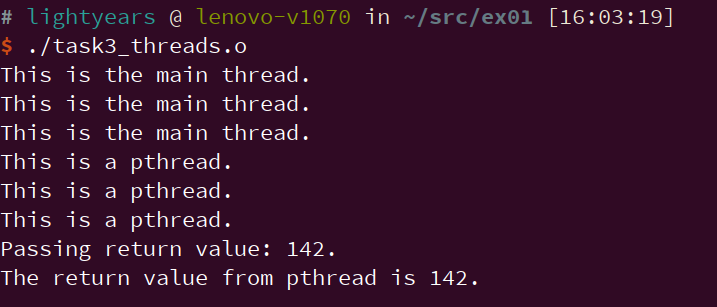
}

主线程中使用pthread\_create创建了一个子线程，随后向控制台打印一些内容，使用pthread\_join等待子线程结束，收集子线程的返回值并打印，最后结束程序。

子线程创建后，也向控制台打印一些内容，随后使用pthread\_exit结束线程。出于演示的目的，此处子进程随机返回一个值。

主线程与子线程同时在系统中运行，“同时”向控制台打印内容。重复运行程序，程序的输出不一定相同。





观察两次运行程序的输出即可发现，多次运行程序的输出不一定相同。

需要注意的是，使用pthread\_exit向主线程返回值时，不能返回指向局部变量的指针，否则子线程结束后，这部分空间被系统回收导致主进程也无法取得子进程的返回值。一个错误的例子如下所示：

*// task3\_threads\_badret.c*

#include <stdlib.h>

#include <stdio.h>

#include <pthread.h>

void thread(void)

{

    int i;

    for (i = 0; i < 3; i++) {

        printf("This is a pthread.\n");

    }

    srand(time(NULL));

    int retval = rand() % 256;

    printf("Passing return value: %d, address = %p.\n", retval, &retval);

    pthread\_exit((void \*)&retval);

}

int main(void)

{

    pthread\_t id;

    if (pthread\_create(&id, NULL, (void \*)thread, NULL) != 0)

    {

        printf("Fail to create pthreads!\n");

        exit(1);

    }

    for (int i = 0; i < 3; i++) {

        printf("This is the main thread.\n");

    }

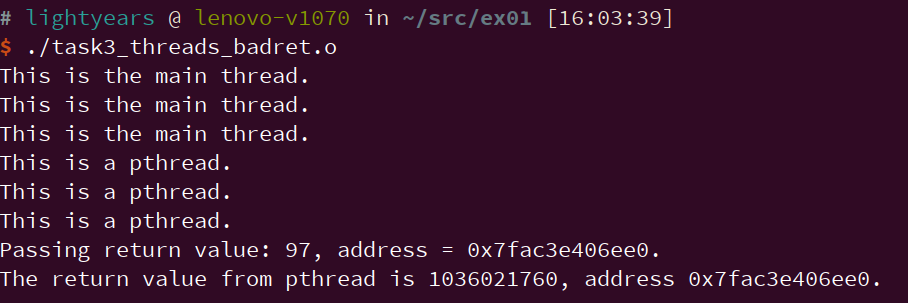
    int \*retval = NULL;

    pthread\_join(id, (void \*\*)&retval);

    printf("The return value from pthread is %d, address %p.\n", \*retval, retval);

}

一个错误的运行结果如下。



1. **利用信号量机制控制多线程的运行顺序，并实现多线程中数据的共享。**

*// task4\_semaphore.c*

#include <stdio.h>

#include <stdbool.h>

#include <time.h>

#include <pthread.h>

#include <semaphore.h>

#include <errno.h>

#include <string.h>

#define QUEUE\_SIZE 3

#define NEXT\_INDEX(INDEX) (((INDEX) + 1) % QUEUE\_SIZE)

int queue[QUEUE\_SIZE][2];

sem\_t empty\_count;

sem\_t filled\_count;

sem\_t reader\_mutex;

sem\_t handler\_mutex;

int alive\_reader\_count;

int reader\_index;

int handler\_index;

void Reader(void \*dataSrc)

{

    FILE \*fp = fopen((const char\*)dataSrc, "r");

    while (true)

    {

        sem\_wait(&empty\_count);

        sem\_wait(&reader\_mutex);

        if (fscanf(fp, "%d %d", &queue[reader\_index][0], &queue[reader\_index][1]) == -1) {

            break;

        }

        reader\_index = NEXT\_INDEX(reader\_index);

        sem\_post(&filled\_count);

        sem\_post(&reader\_mutex);

    }

    --alive\_reader\_count;

    sem\_post(&reader\_mutex);

    fclose(fp);

}

void Handler(void \*operator)

{

    while (alive\_reader\_count > 0)

    {

        struct timespec timeout;

        clock\_gettime(CLOCK\_REALTIME, &timeout);

        timeout.tv\_sec += 1;

        if (sem\_timedwait(&filled\_count, &timeout) == -1) {

            continue;

        }

        sem\_wait(&handler\_mutex);

        int a = queue[handler\_index][0];

        int b = queue[handler\_index][1];

        queue[handler\_index][0] = 0;

        queue[handler\_index][1] = 0;

        int result;

        switch (((const char\*)operator)[0])

        {

        case '+': result = a + b; break;

        case '\*': result = a \* b; break;

        }

        printf("[%s] %2d %s %2d = %2d\n", (const char\*)operator, a, (const char\*)operator, b, result);

        handler\_index = NEXT\_INDEX(handler\_index);

        sem\_post(&empty\_count);

        sem\_post(&handler\_mutex);

    }

}

int main(void)

{

    pthread\_t reader1, reader2, handler1, handler2;

    sem\_init(&reader\_mutex, 0, 1);

    sem\_init(&handler\_mutex, 0, 1);

    sem\_init(&empty\_count, 0, QUEUE\_SIZE);

    sem\_init(&filled\_count, 0, 0);

    pthread\_create(&reader1, NULL, (void \*)Reader, "task4\_data1.txt");

    pthread\_create(&reader2, NULL, (void \*)Reader, "task4\_data2.txt");

    alive\_reader\_count = 2;

    pthread\_create(&handler1, NULL, (void \*)Handler, "+");

    pthread\_create(&handler2, NULL, (void \*)Handler, "\*");

    pthread\_join(reader1, NULL);

    pthread\_join(reader1, NULL);

    pthread\_join(handler1, NULL);

    pthread\_join(handler2, NULL);

}

上述代码中以生产者消费者模型为例，演示信号量的使用。主线程中创建了两个消费者线程Handler和两个生产者线程Reader。两个Reader线程都从文件中读取数字，并放到环形缓存queue中。文件中预设数字1~10以及-1~-10。两个消费者从环形缓存queue中取得数字，并对取得的数字做加法或乘法操作。

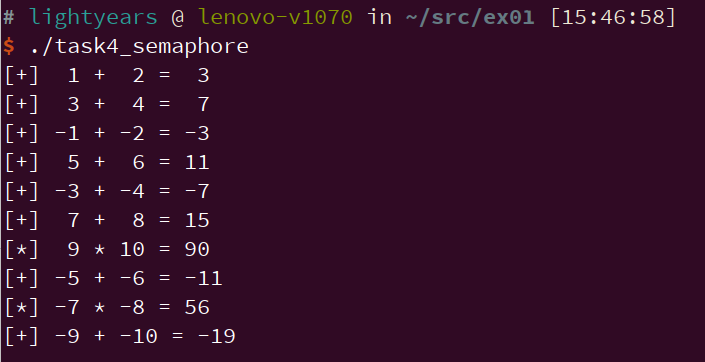
代码中设有empty\_count、filled\_count、reader\_mutex、handler\_mutex四个信号量。

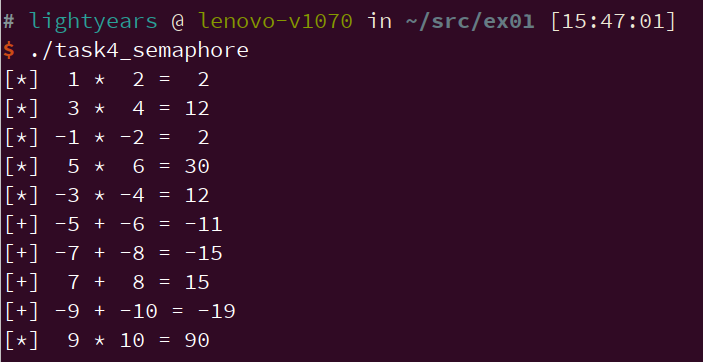
前两个信号量empty\_count和filled\_count分别表示环形缓存中空的储藏单元（已经被handler处理，可被reader存放新的数据）和满的储藏单元（被reader存放从文件中读取的数据，而还未被handler处理）的数量，初始值分别设为环形缓存的容量和0。生产者向环形缓存存入数据时，需要使用sem\_wait(empty\_count)先取得一个空单元，存放数据后使用sem\_post(filled\_count)增加一个满单元。消费者从环形缓存取得数据时，则是先sem\_wait(filled\_count)，然后sem\_post(empty\_count)。

reader\_mutex初始值为1，相当于互斥锁。当生产者线程需要操作生产者相关的变量时，需要使用sem\_wait取得reader\_mutext，操作结束后再使用sem\_post释放锁。handler\_mutex的作用类似于reader\_mutext，初始值也为1。

代码中使用alive\_reader\_count保存活跃的生产者数量，初始值为2。当生产者读取到文件的末尾再无数据可读时，这个生产者线程将退出，在退出前时alive\_reader\_count减一。当消费者处理完环形缓存中的数据且alive\_reader\_count等于0时，消费者线程退出。当所有的生产者和消费者线程退出后，程序结束。

两次运行程序的输出如下所示。可见重复运行程序，程序的输出不一定相同。





1. **分析Linux系统下多进程与多线程中的区别。**

对实验题1和实验题3中的程序稍作修改，可得到以下两个程序。

*// task5\_fork.c*

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

int ref = 1;

int subprocess(void) {

    for (int i = 0; i < 3; ++i) {

        printf("Child process: ref = %d.\n", ref++);

    }

    exit(0);

}

int main(void) {

    if (fork() == 0) {

        subprocess();

    }

    for (int i = 0; i < 3; ++i) {

        printf("Parent Process: ref = %d.\n", ref++);

    }

    wait(NULL);

}

*// task5\_pthread.c*

#include <stdlib.h>

#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

int ref = 1;

void subthread(void)

{

    for (int i = 0; i < 3; ++i) {

        printf("Sub thread: ref = %d.\n", ref++);

    }

}

int main(void)

{

    pthread\_t id;

    pthread\_create(&id, NULL, (void \*)subthread, NULL);

    for (int i = 0; i < 3; ++i) {

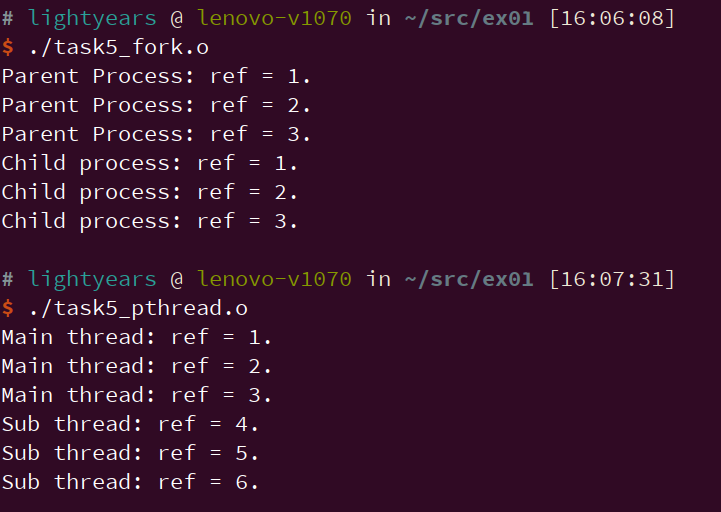
        printf("Main thread: ref = %d.\n", ref++);

    }

    pthread\_join(id, NULL);

}

编译并运行程序，得到以下结果：



程序中ref是一个C语言全局变量，观察程序的运行结果可以发现，多进程中每个进程拥有各自的全局变量副本，对自己进程的全局变量进行的修改不会影响其他进程；而多线程中，所有的线程共享一个全局变量，其中一个线程对全局变量的修改会影响所有线程。由于多个线程之间共享量大量信息，一般认为使用多线程的系统资源开销小于使用多进程。

可通过以下程序直观地比较创建1000个子进程与创建1000个线程所需要花费的时间。

*// demo1\_fork.c*

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

int count = 0;

int subprocess(void) {

    if (count < 1000) {

        if (fork() == 0) {

            count++;

            subprocess();

            exit(0);

        }

        wait(NULL);

    }

    exit(0);

}

int main(void) {

    subprocess();

    wait(NULL);

}

*// demo1\_pthread.c*

#include <stdlib.h>

#include <stdio.h>

#include <pthread.h>

#include <unistd.h>

int count = 0;

void subthread(void)

{

    pthread\_t id;

    if (count < 1000) {

        count++;

        pthread\_create(&id, NULL, (void \*)subthread, NULL);

        pthread\_join(id, NULL);

    }

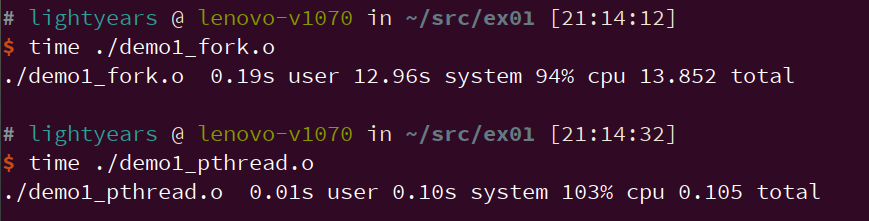
}

int main(void)

{

    subthread();

}



通过上面的实验结果可看出fork的时间开销确实多于创建线程。

更具体地说，使用fork创建子进程时，子进程是得到父进程所有资源的副本；而多线程则是各个线程共享资源。在进程间进行信息交换需要操作系统提供的进程间通信机制，如操作系统实验中的管道、信号、消息队列等机制。而在多线程中对共享内存的修改则需要小心翼翼地使用如实验题4中提到的信号量等工具，以避免竞争冒险。

1. **用fork写一个测试程序，从父进程和子进程中打印出各自的PID，并把wait(), exit()系统调用加进去，使子进程返回退出状态给父进程，并将它包含在父进程的打印信息中。对运行结果进行分析。**

*// task6.c*

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

int main(void) {

    int pid = -1;

    pid = fork();

    switch (pid)

    {

    case -1:

        printf("Fail to fork.");

        break;

    case 0:

        printf("This is the child process, pid = %d.\n", getpid());

        exit(EXIT\_FAILURE);

        break;

    default:

        printf("This is the parent process, pid = %d.\n", getpid());

        int wstatus;

        wait(&wstatus);

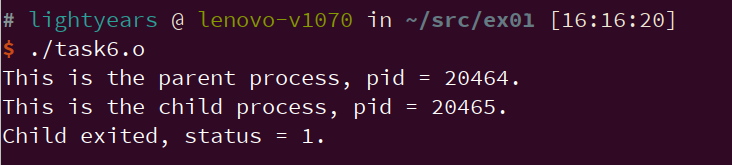
        printf("Child exited, status = %d.\n", WEXITSTATUS(wstatus));

        break;

    }

}

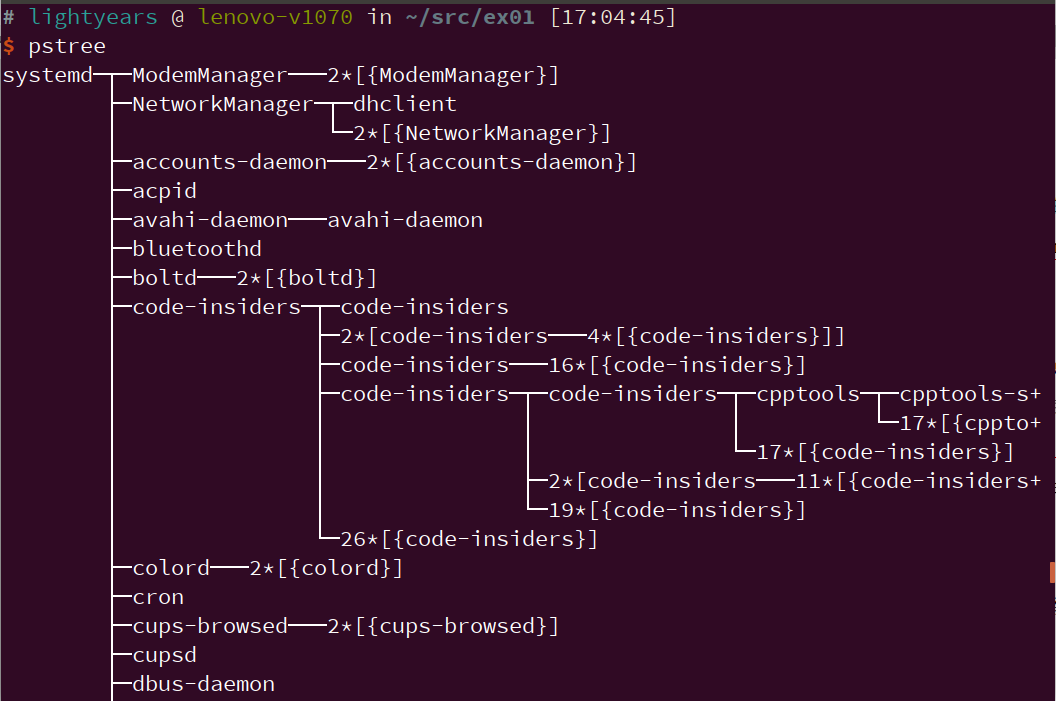
在实验题1的基础上不难写出上面的代码。在进程中，可以使用getpid获得当前进程的PID。在父进程收集子进程的返回值时，需要使用WEXITSTATUS宏。程序的一次运行结果如下图所示。



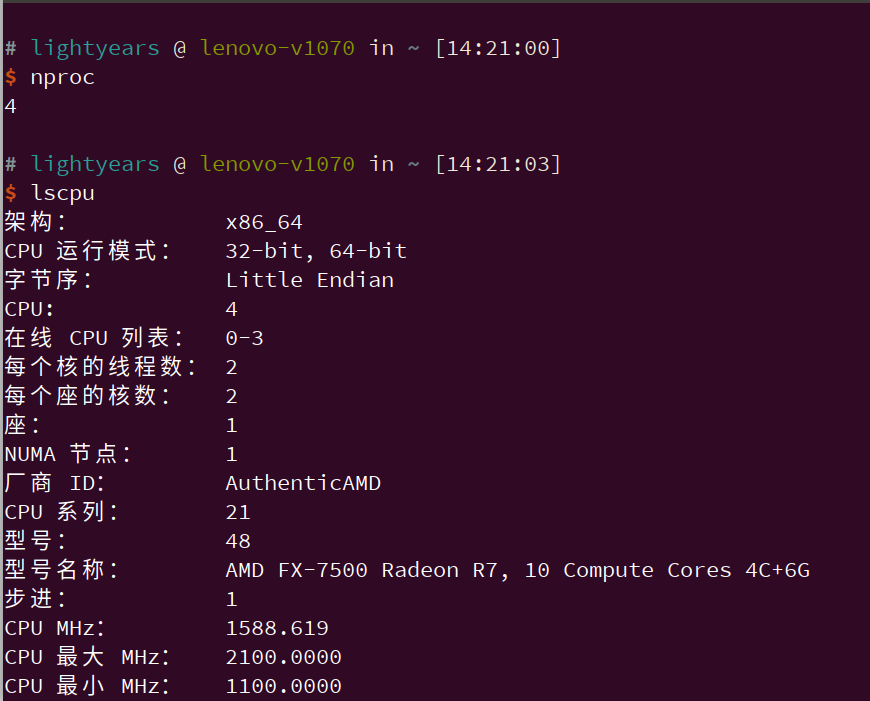
1. **用代码编辑器查看自己计算机上的内核源代码中的include/linux/sched.h文件中的task\_struct结构，列出类型为list\_head的字段，并说明哪几个字段可以形成进程的树结构，通过pstree命令查看自己机器上的进程树结构。**

Struct list\_head只有两个字段：next和prev，分别指向链表的后继节点和前驱节点，用于创建双向循环链表结构。Struct task\_struct中，real\_parent、parent、children和sibling字段可以形成进程的家族结构。Real\_parent和parent字段分别指向当前进程的父进程task\_struct以及养父（接受SIGCHLD信号的进程）的task\_struct。Children和sibling字段则以struct list\_head形式与当前进程的孩子进程和兄弟进程task\_struct构成双向循环链表。

Pstree命令的一次运行结果如下图所示：

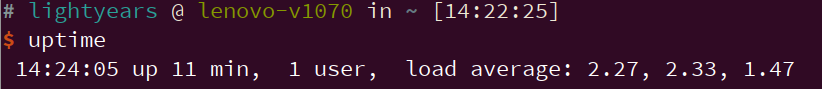


1. **在Linux中用nproc或者lscpu 命令来查看CPU核心数，多个核心数就意味着系统可以在同一时间处理多个进程。**

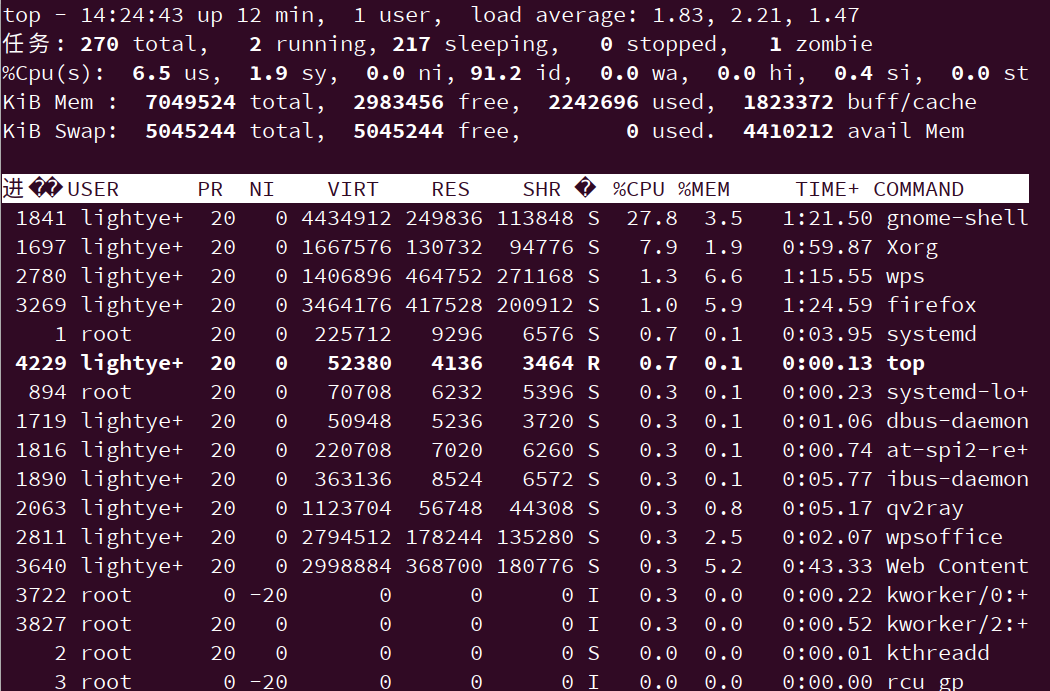


Nproc给出机器上的CPU有4个处理单元。通过观察lscpu命令的输出可知，实验所用的机器CPU有4个核心（与nproc给出的结果一致），每个核心能同时运行两个线程。因此机器能同时执行8个线程。

1. **在Linux中用top或者uptime 命令来查看系统负载情况，返回的load average的3个数字表示的系统在1分钟、5分钟、15分钟内平均负载情况。**



通过uptime的输出可知，实验所用的机器上登录了1个用户，过去1分钟、5分中和15分钟内的系统平均负载分别为2.27、2.33和1.57。



通过top命令，不仅能得到当前系统的平均负载信息，还能获得系统中当前运行进程的概览。如图所示，当前系统中占用CPU最多的进程为gnome-shell、Xorg和wps。

## 实验思考题

1. **多进程并发执行时各个进程的内存分配情况如何？如何监测其分配情况？**

多进程在并发执行时，可以通过pmap命令监测各个进程的内存分配情况。编写并运行以下示例代码。

*// task7.c*

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <unistd.h>

int main(void) {

    int pid = -1;

    pid = fork();

    switch (pid)

    {

    case -1:

        printf("Fail to fork.");

        break;

    case 0:

        printf("This is the child process, pid = %d.\n", getpid());

        sleep(20);

        break;

    default:

        printf("This is the parent process, pid = %d.\n", getpid());

        sleep(1);

        printf("Sleep for 20 seconds...\n");

        sleep(20);

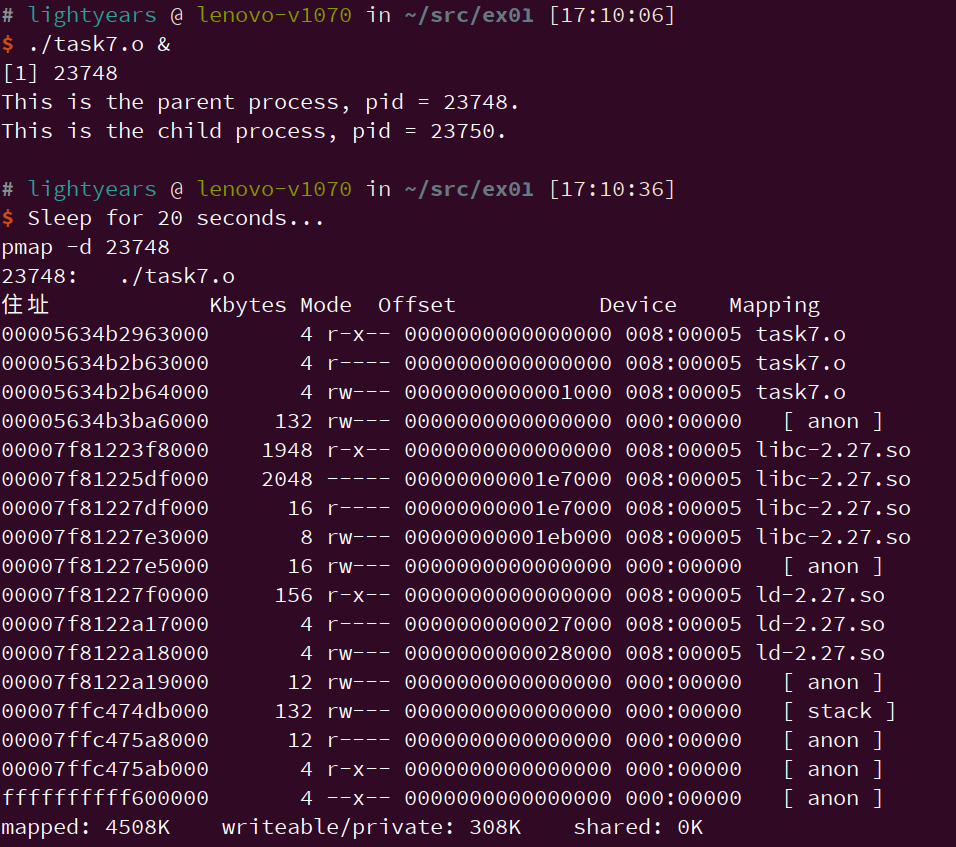
        wait(NULL);

        break;

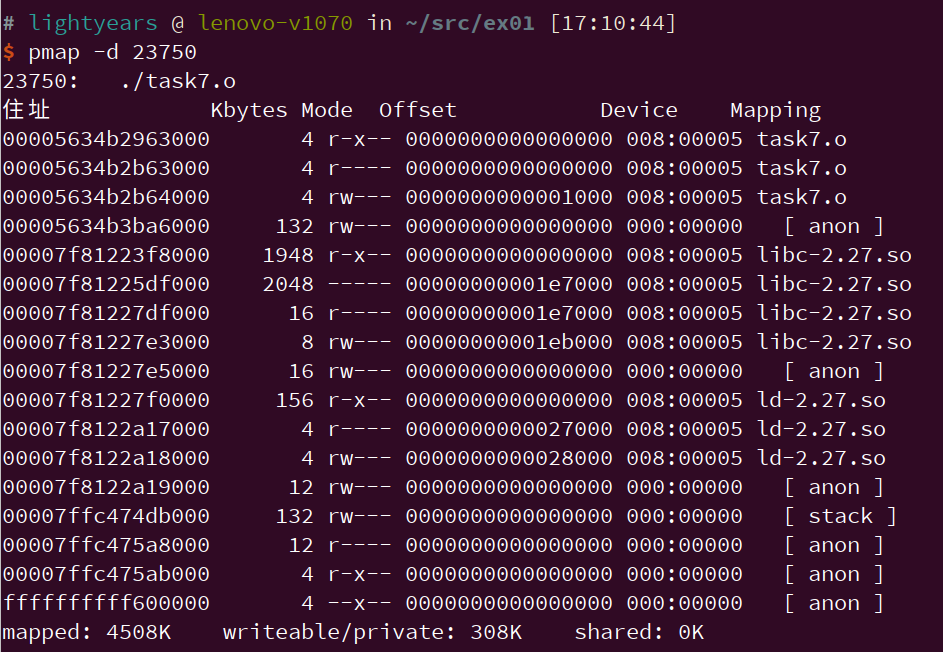
    }

}

运行程序，并使用pmap分别查看父进程和子进程的内存分配情况。下图为父进程内存分配情况。



下图为子进程内存分配情况。



Pmap指令能报告进程内存的使用量和内存映射关系。在报告的最后一行中，mapped表示该进程映射的虚拟地址空间大小，也就是该进程预先分配的虚拟内存大小；writeable/private表示进程所占用的私有地址空间大小，即进程实际使用的内存大小；shared表示进程和其他进程共享的内存大小。

1. **多线程中数据是如何共享的？**

多线程中数据通过共享内存的方式共享，例如所有线程都能修改C语言的全局变量。但修改共享内存时，要注意使用信号量等同步机制，以避免多个进程对同一变量同时进行修改所带来的数据冒险问题。

1. **Linux系统下多进程与多线程中的区别是什么？**

根据实验题5可做出以下解答：简要地说，多进程中每个进程拥有各自的全局变量副本，对自己进程的全局变量进行的修改不会影响其他进程；而多线程中，所有的线程共享一个全局变量，其中一个线程对全局变量的修改会影响所有线程。由于多个线程之间共享量大量信息，一般认为使用多线程的系统资源开销小于使用多进程。

更具体地说，使用fork创建子进程时，子进程是得到父进程所有资源的副本；而多线程则是各个线程共享资源。在进程间进行信息交换需要操作系统提供的进程间通信机制，如操作系统实验中的管道、信号、消息队列等机制。而在多线程中对共享内存的修改则需要小心翼翼地使用如实验题4中提到的信号量等工具，以避免数据的竞争冒险。