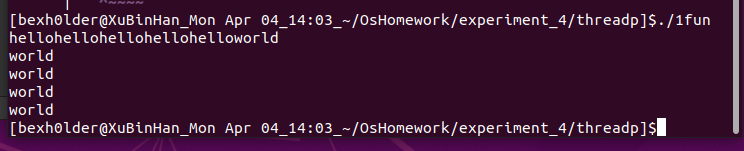
# 实验四 线程的创建和简单控制

1. 调试并运行给出的”hello world”单线程程序，观察输出结果，并分析原因

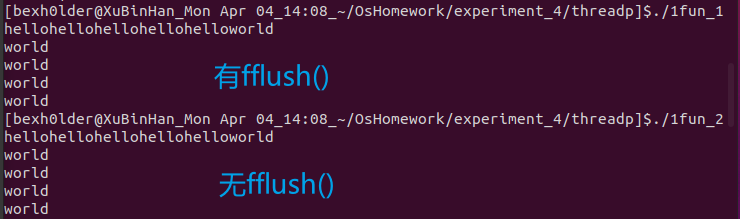
编译链接通过后，多次运行例程，观察进程并发执行结果

1. 此例程使用函数调用，理解函数调用是顺序执行的;



**程序运行后先输出5个hello再输出5个word，p\_msg("hello");在p\_msg("world\n"); 之前，先调用输出hello，再调用输出world，与执行顺序对应，顺序执行**

1. 理解fflush()的作用



**有无fflush()函数的输出并无变化，区别在于没有fflush()时，五个hello和一个world是在程序运行一段时间后瞬间输出，而之后的四个world是过一秒输出一个。**

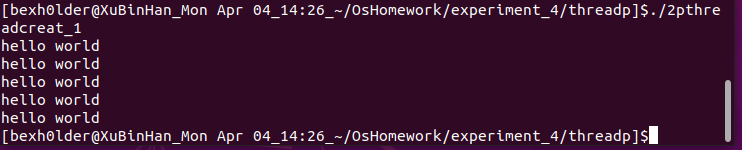
**fflush(stdout)刷新标准输出缓冲区，把输出缓冲区里的东西强制打印到标准输出设备上。程序的输出内容一般不会立即输出，而是在程序结束后再输出。fflush(stdout)会强制每次printf()都立即显示在标准输出设备上。stdout默认是是行缓冲的，遇到 \n 就写内容清缓存，而加上fflush(stdout); 与有\n作用一样，只是不换行。**

1. 如果希望一个程序能同时并发执行很多函数，它可以通过创建多个线程实现，只要将上面的程序做两个小的改动，即可让程序并发执行两个函数，下面的例程是一个简单的”hello world”多线程程序

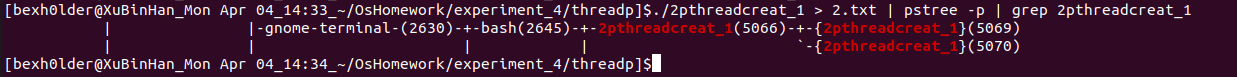
编译链接通过后，多次运行例程，观察进程和线程并发执行结果

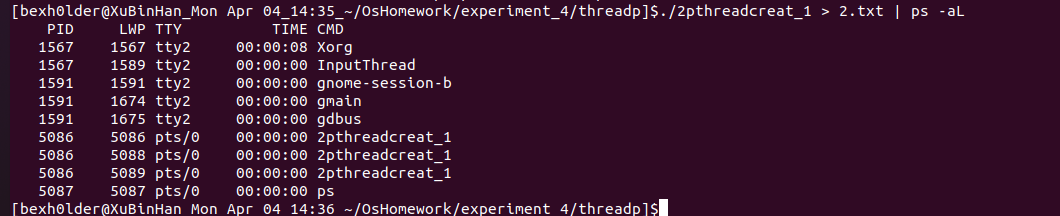
需要使用gcc -o 2pthreadcreat\_1 -pthread 2pthreadcreat.c命令来编译链接

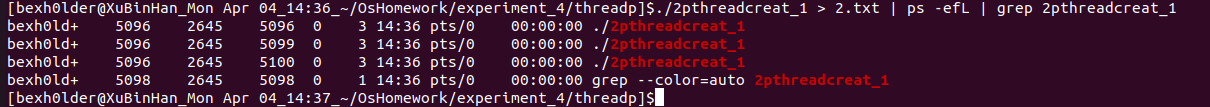
1. 从输出内容，观察线程的并发执行



1. 运行例程，用ps -aL、ps -efL和pstree -p观察线程信息







1. 理解pthread\_create()函数作用

**pthread\_create()作用是创建新线程（就是确定调用该线程函数的入口点）。在线程 创建以后（即调用pthread\_create()后），就开始运行相关的线程函数**

**int pthread\_create(pthread\_t \*thread,**

**const pthread\_attr\_t \*attr,**

**void \*(\*start\_routine) (void \*),**

**void \*arg);**

**各个参数的含义是：**

1. **pthread\_t \*thread：传递一个 pthread\_t 类型的指针变量，也可以直接传递 某个pthread\_t 类型变量的地址。pthread\_t 是一种用于表示线程的数据类型， 每一个 pthread\_t 类型的变量都可以表示一个线程。**

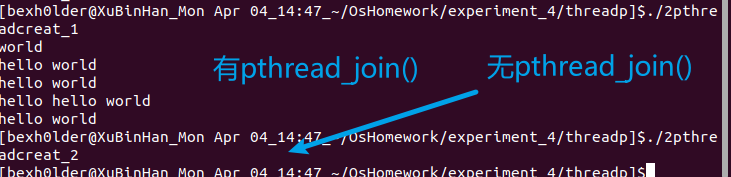
**b. const pthread\_attr\_t \*attr：用于手动设置新建线程的属性，例如线程的调用 策略、线程所能使用的栈内存的大小等。大部分场景中，我们都不需要手动修 改线程的属性，将 attr 参数赋值为 NULL，pthread\_create() 函数会采用系统默 认的属 性值创建线程。**

**pthread\_attr\_t 类型以结构体的形式定义在<pthread.h>头文件中，此类型的变量专 门表示线程的属性，NULL表示使用默认属性**

**c. void \*(\*start\_routine) (void \*)：以函数指针的方式指明新建线程需要执行的函 数，该函数的参数最多有 1 个（可以省略不写），形参和返回值的类型都必须为 void\* 类型。void\* 类型又称空指针类型，表明指针所指数据的类型是未知的。 使用此类型指针时，我们通常需要先对其进行强制类型转换，然后才能正常访 问指针指向的数据。**

**d. void \*arg：指定传递给 start\_routine 函数的实参，当不需要传递任何数据时， 将arg赋值为 NULL 即可。**

1. 理解pthread\_join()函数作用。尝试将pthread\_join()删除，观察运行结果的变化



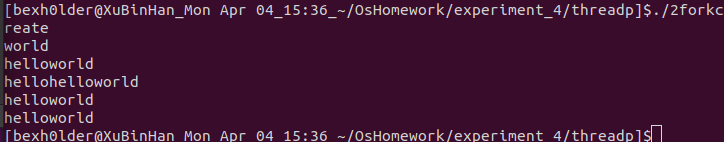
**pthread\_join()的作用是以阻塞的方式等待thread指定的线程结束，当函数返回时，被等待线程的资源被收回，如果线程已经结束，那么该函数会立即返回，并且thread指定的线程必须是joinable的**

**int pthread\_join(pthread\_t \*thread,**

**void \*\*retval);**

**各个参数的含义是：**

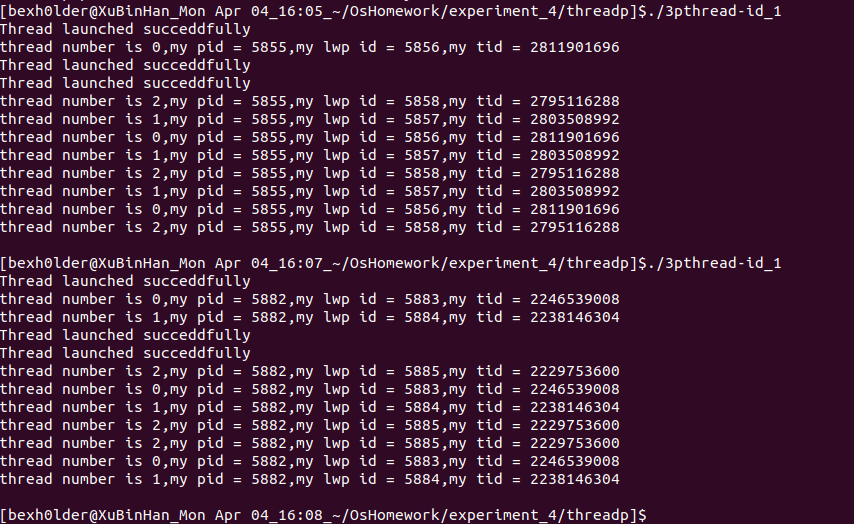
1. **thread:线程标识符，即线程ID，标识唯一线程；**
2. **retval:用户定义的指针，用来存储被等待线程的返回值**
3. 改写代码，使同样的输出内容由并发的父子进程实现，体会进程和线程概念的差异
4. *//"hello world"多进程程序*
5. #include <stdio.h>
6. #include <stdlib.h>
7. #include <unistd.h>
8. #include <sys/wait.h>
9. int main()
10. {
11. int pid;
12. int i;
13. if((pid=fork())==0)
14. {                                         */\*这里是子进程执行的代码\*/*
16. for(i = 0 ; i < 5 ; i ++)
17. {
18. printf("%s","hello");
19. fflush(stdout);*//清除文件缓冲区，文件以写方式打开时将缓冲区内容写入文件*
20. sleep(1);
21. }
22. exit(0);
23. }
24. else
25. {                                         */\*这里是父进程执行的代码\*/*
26. *//wait(0);                            /\*如果此处没有这一句会如何？\*/*
27. for(i = 0 ; i < 5 ; i ++)
28. {
29. printf("%s","world\n");
30. fflush(stdout);*//清除文件缓冲区，文件以写方式打开时将缓冲区内容写入文件*
31. sleep(1);
32. }
33. }
34. return 0;
35. }



1. 运行程序，观察线程并发执行的效果，以及关于线程的各种ID

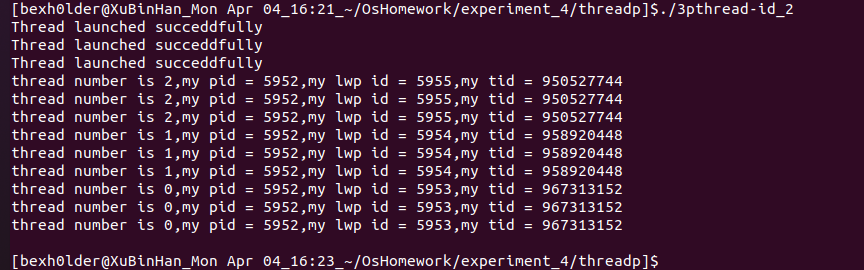
**gcc -o 3pthread-id\_1 -pthread 3pthread-id.c**

1. 分析输出内容的顺序不确定的原因

****

**执行过程中调用了sleep()函数，执行后线程阻塞，资源释放，使别的线程能够执 行，从而导致顺序不固定**

1. 删除sleep(1);观察输出内容的变化，并分析原因



**删除sleep(1)后，顺序执行线程，一个线程执行完之后再执行最后一个进程，先执 行main函数里的代码创建了三个线程，而后执行完调用过程后执行线程，最后被 创建了线程最先被执行**

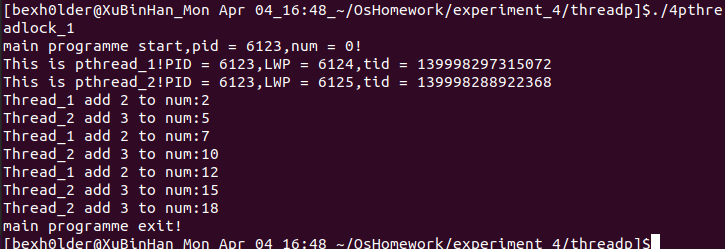
1. 分析输出内容，理解线程各种id值的内容

**pid: 进程ID。**

**lwp: 线程ID。在用户态的命令(比如ps)中常用的显示方式。**

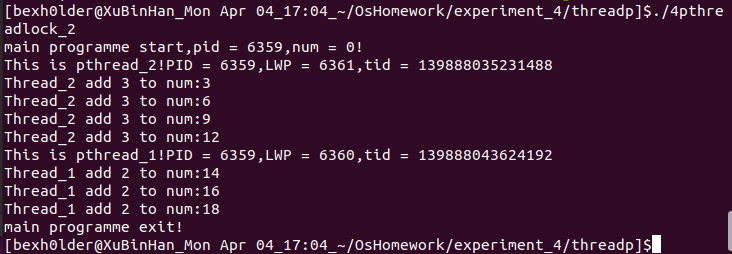
**tid: 线程ID，等于lwp。tid在系统提供的接口函数中更常用，比如syscall(SYS\_gettid) 和syscall(\_\_NR\_gettid)。**

1. 运行下述例程，理解POSIX互斥锁的简单使用
2. 保留注释，运行程序，可观察到输出混乱。请分析出现混乱的原因

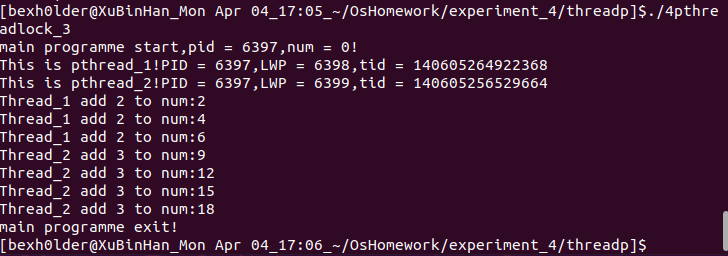


**执行过程中调用了sleep()函数，执行后线程阻塞，资源释放，使别的线程能够执 行，从而导致输出混乱**

1. 删除sleep函数，运行程序，可观察到输出混乱现象消失了。思考这样是否从根本上彻底解决问题



1. 恢复sleep函数，删除注释，启用互斥锁，观察运行结果的变化，理解互斥锁的作用



保证在一个线程调用临界资源的时候另一个进程不会调用相同临界资源

1. 掌握互斥锁的基本操作：初始化，加锁（P操作）和解锁（V操作）

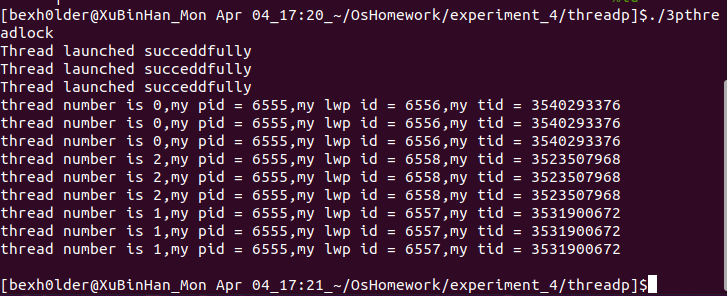
加锁：使临界资源无法再被其他线程调用

解锁：使临界资源可以被其他线程调用

1. 各线程输出内容不再混乱，和删除sleep函数效果类似，但是原因不同，请分析各自原因

删除sleep函数是能够使线程没有等待的时间，持续执行，直到完全执行完线程2 之后再执行线程1，而互斥锁是将资源锁定，只要资源没解锁，其他的线程就不能 调用该资源所以会先执行完线程1再执行线程2

1. 修改例程3，使用互斥锁，使各线程输出内容不交错
2. #include <stdio.h>
3. #include <stdlib.h>
4. *//#include <sys/types.h>*
5. #include <sys/syscall.h>
6. #include <pthread.h>
7. *//#include <unistd.h>*
8. void \*PrintThread(void \* );
9. */\*互斥量 \*/*
10. pthread\_mutex\_t mutex;
11. #define Num\_Threads 3
12. int main()
13. {
14. int i,ret;
15. pthread\_t a\_thread;
16. int ThdNum[Num\_Threads];
17. for (i = 0; i < Num\_Threads; i++)
18. ThdNum[i] = i;
19. */\*互斥初始化\*/*
20. pthread\_mutex\_init (&mutex, NULL);
21. for(i = 0; i < Num\_Threads;i++)
22. {
23. ret = pthread\_create(&a\_thread,NULL,PrintThread,(void \*)&ThdNum[i]);
24. if (ret == 0)
25. printf("Thread launched succeddfully\n");
26. }
27. i = getchar();
28. return (0);
29. }
30. void \*PrintThread(void \* num)
31. {
32. int i;
33. for(i = 0;i < 3; i++)
34. {
35. pthread\_mutex\_lock(&mutex); */\*获取互斥锁\*/*
36. */\*临界资源\*/*
37. printf("thread number is %d,my pid = %d,my lwp id = %d,my tid = %u\n",\*((int \*)num),getpid(),syscall(SYS\_gettid),pthread\_self());
38. *//sleep(1);*
39. pthread\_mutex\_unlock(&mutex); */\*释放互斥锁\*/*
40. }
41. return NULL;
42. }



1. 使用POSIX互斥锁，保证运行结果获得期望值

观察程序加锁前后变化，理解互斥锁使用

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <pthread.h>
4. #include <errno.h>
5. #include <unistd.h>
6. */\*互斥量 \*/*
7. pthread\_mutex\_t mutex;
8. int myglobal=0;
9. void \*thread\_function(void \*arg)
10. {
11. int i,j;
12. for(i = 0; i <20; i++){
13. pthread\_mutex\_lock(&mutex); */\*获取互斥锁\*/*
14. j = myglobal;
15. j++;
16. printf(".");
17. fflush(stdout);
18. usleep(1000);
19. myglobal = j;
20. pthread\_mutex\_unlock(&mutex); */\*释放互斥锁\*/*
21. }
22. return NULL;
23. }
24. int main(){
25. pthread\_t mythread;
26. int i;
27. */\*互斥初始化\*/*
28. pthread\_mutex\_init (&mutex, NULL);
29. if( pthread\_create(&mythread,NULL,thread\_function,NULL))
30. {
31. printf("error creating thread!\n");
32. abort();
33. }
34. for(i = 0; i <20; i++)
35. {
36. pthread\_mutex\_lock(&mutex); */\*获取互斥锁\*/*
37. myglobal++;
38. printf("O");
39. fflush(stdout);
40. usleep(1000);
41. pthread\_mutex\_unlock(&mutex); */\*释放互斥锁\*/*
42. }
43. if( pthread\_join(mythread,NULL))
44. {
45. printf("error joining thread!\n");
46. abort();
47. }
48. printf("\nmyglobal equald %d\n",myglobal);
49. exit(0);
50. }

