线程调度的几个基本知识点

多线程并发执行时有很多同学捋不清楚调度的随机性会导致哪些问题，要知道如果访问临界资源不加锁会导致一些突发情况发生甚至死锁。

关于线程调度，需要深刻了解以下几个基础知识点：

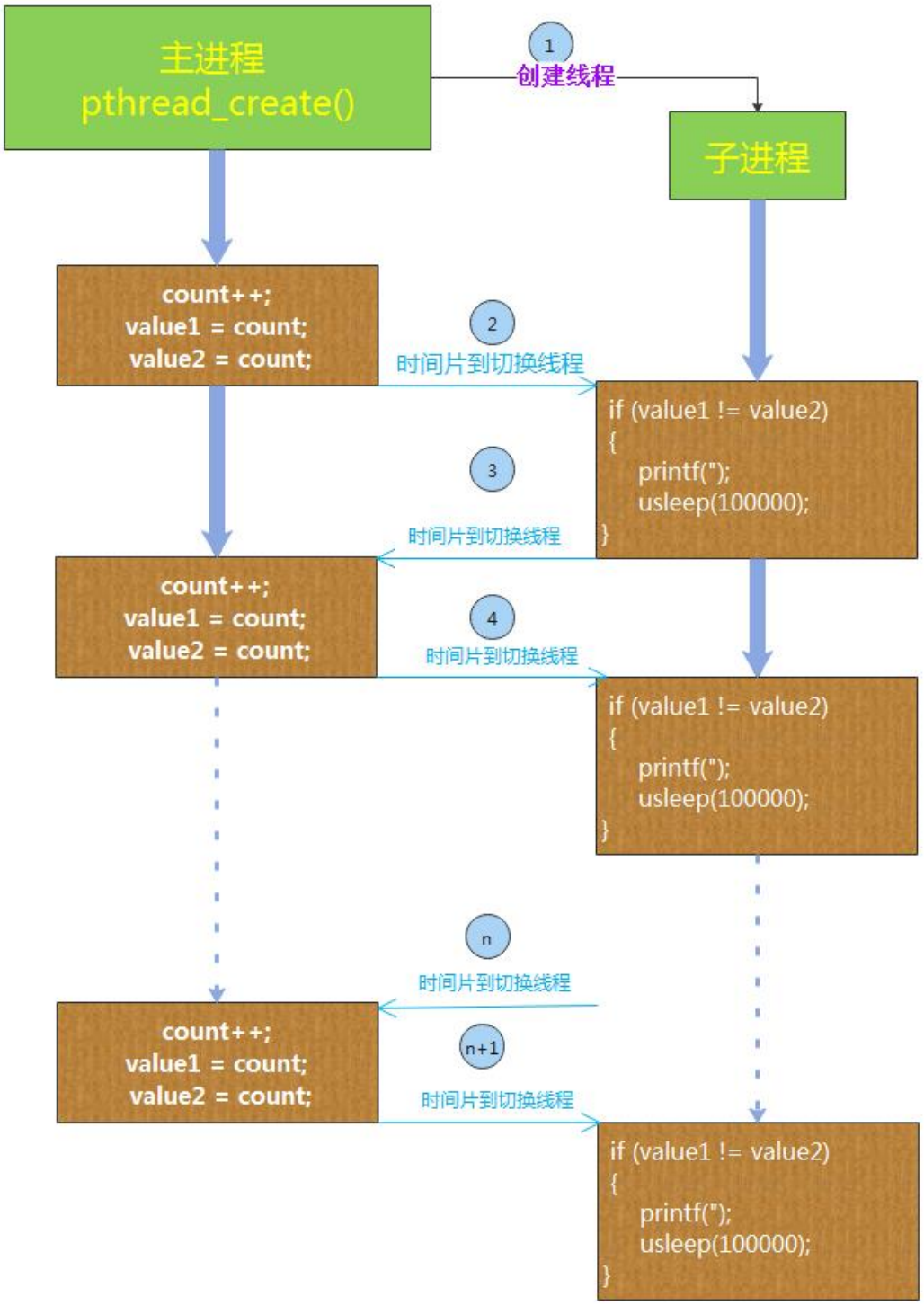
1. 调度的最小单位是轻量级进程【比如我们编写的 hello world 最简单的 C 程序，执行时就是一个轻量级进程】或者线程；
2. 每个线程都会分配一个时间片，时间片到了就会执行下一个线程；
3. 线程的调度有一定的随机性，无法确定什么时候会调度；
4. 在同一个进程内，创建的所有线程除了线程内部创建的局部资源，进程创建的其他资源所有线程共享；比如：主线程和子线程都可以访问全局变量，打开的文件描述符等。

实例

再多的理论不如一个形象的例子来的直接。

预期代码时序

假定我们要实现一个多线程的实例，预期程序执行时序如下：



期待时序

期待的功能时序：

1. 主进程创建子线程，子线程函数 function（）；
2. 主线程 count 自加，并分别赋值给 value1,value2；

3. 时间片到了后切换到子线程，子线程判断 value1、value2 值是否相同，如果不同就打印信息 value1,value2,count 的值，但是因为主线程将 count 先后赋值给了 value1,value2，所以 value1,value2 的值应该永远相同，所以不应该打印任何内容；
4. 重复 2、3 步骤。

代码 1

好了，现在我们按照这个时序编写代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <unistd.h>

unsigned int value1,value2, count=0;
void *function(void *arg);
int main(int argc,  char *argv[])
{
    pthread_t  a_thread;

    if (pthread_create(&a_thread, NULL, function, NULL) < 0)
    {
        perror("fail to pthread_create");
        exit(-1);
    }
    while ( 1 )
    {
        count++;
        value1 = count;
        value2 = count;
    }
    return 0;
}

void  *function(void *arg)
{
    while ( 1 )
    {
        if (value1 != value2)
        {
```

```
33              printf("count=%d , value1=%d, value2=%d\n",  count, value1, value2)
;
34              usleep(100000);
35          }
36      }
37      return  NULL;
38  }
```

乍一看，该程序应该可以满足我们的需要，并且程序运行的时候不应该打印任何内容，但是实际运行结果出乎我们意料。

编译运行：

```
gcc test.c -o run -lpthread
./run
```

代码 1 执行结果

执行结果：　　可以看到子程序会随机打印一些信息，为什么还有这个执行结果呢？其实原因很简单，就是我们文章开头所说的，线程调度具有碇随机性，我们无法规定让内核何时调度某个线程。有打印信息，那么这说明此时 value1 和 value2 的值是不同的，那也说明了调度子线程的时候，是在主线程向 value1 和 value2 之间的位置调度的。

代码 1 执行的实际时序

实际上代码的执行时序如下所示：

如上图，在某一时刻，当程序走到 value2 = count;这个位置的时候，内核对线程进行了调度，于是子进程在判断 value1 和 value2 的值的时候，发现这两个变量值不相同，就有了打印信息。

该程序在下面这两行代码之间调度的几率还是很大的。

```
value1 = count;
value2 = count;
```

解决方法

如何来解决并发导致的程序没有按预期执行的问题呢？对于线程来说，常用的方法有 posix 信号量、互斥锁，条件变量等，下面我们以互斥锁为例，讲解如何避免代码 1 的问题的出现。

互斥锁的定义和初始化：

```
pthread_mutex_t  mutex;
pthread_mutex_init(&mutex, NULL)
```

申请释放锁：

```
pthread_mutex_lock(&mutex);
pthread_mutex_unlock(&mutex);
```

原理：进入临界区之前先申请锁，如果能获得锁就继续往下执行， 如果申请不到，就休眠，直到其他线程释放该锁为止。

代码 2

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>
 4 #include <pthread.h>
 5 #include <unistd.h>
 6 #define _LOCK_
 7 unsigned int value1,value2, count=0;
 8 pthread_mutex_t  mutex;
 9 void *function(void *arg);
10
11 int main(int argc,  char *argv[])
12 {
13     pthread_t  a_thread;
14
15     if (pthread_mutex_init(&mutex, NULL) < 0)
16     {
17         perror("fail to mutex_init");
18         exit(-1);
19     }
```

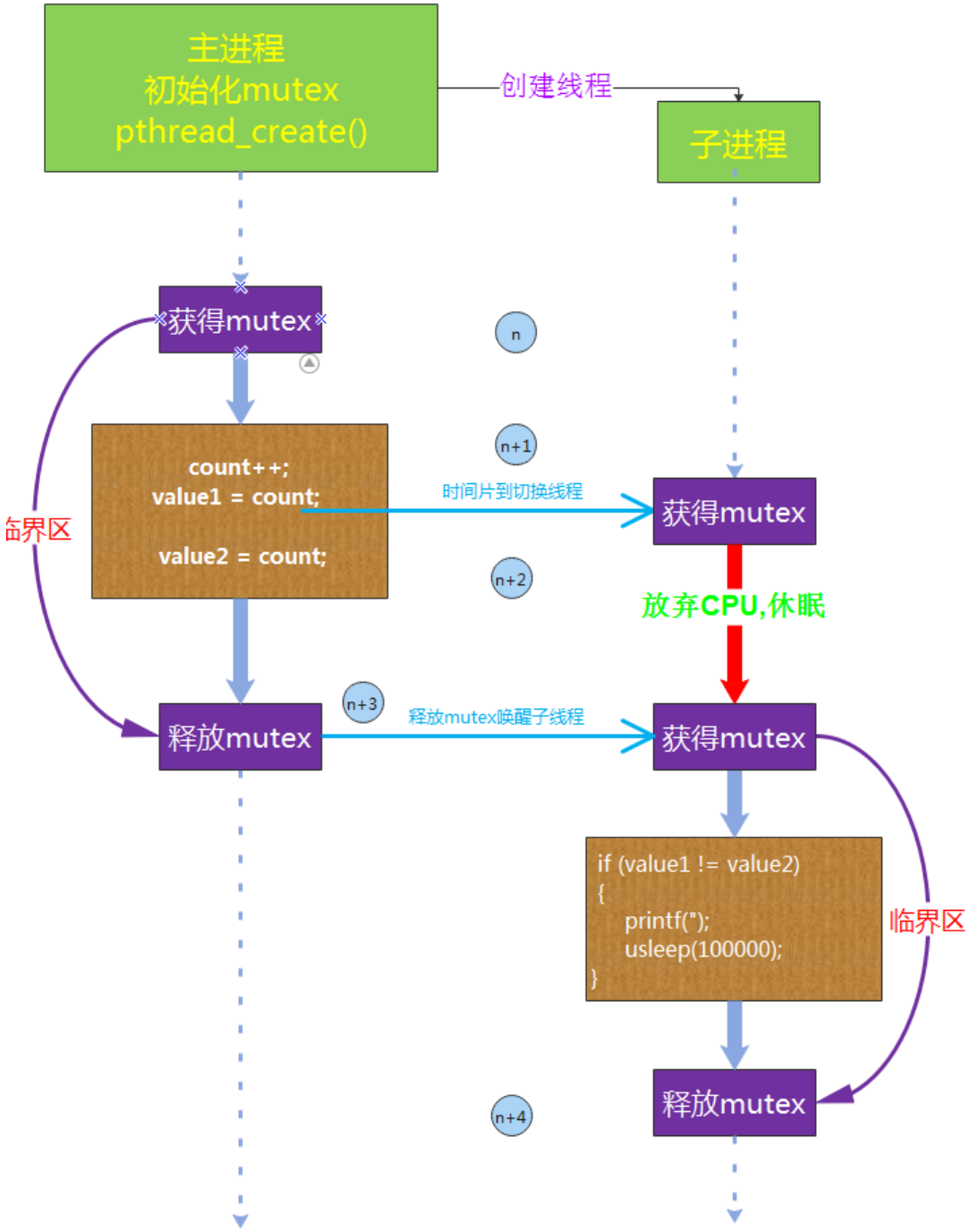
```
20
21      if (pthread_create(&a_thread, NULL, function, NULL) < 0)
22      {
23          perror("fail to pthread_create");
24          exit(-1);
25      }
26      while ( 1 )
27      {
28          count++;
29  #ifdef  _LOCK_
30          pthread_mutex_lock(&mutex);
31  #endif
32          value1 = count;
33          value2 = count;
34  #ifdef  _LOCK_
35          pthread_mutex_unlock(&mutex);
36  #endif
37      }
38      return 0;
39   }
40
41  void  *function(void *arg)
42  {
43       while ( 1 )
44       {
45  #ifdef _LOCK_
46          pthread_mutex_lock(&mutex);
47  #endif
48
49          if (value1 != value2)
50          {
51              printf("count=%d , value1=%d, value2=%d\n",  count, value1, value2)
;
52              usleep(100000);
53          }
54  #ifdef _LOCK_
55          pthread_mutex_unlock(&mutex);
56  #endif
57       }
58      return  NULL;
59   }
```

如上述代码所示:主线程和子线程要访问临界资源 value1,value2 时，都必须先申请锁，获得锁之后才可以访问临界资源，访问完毕再释放互斥锁。该代码执

行之后就不会打印任何信息。我们来看下，如果程序在下述代码之间产生调度时，程序的时序图。

```
value1 = count;
value2 = count;
```

时序图如下：



如上图所示：

1. 时刻 n，主线程获得 mutex，从而进入临界区；
2. 时刻 n+1，时间片到了，切换到子线程；
3. n+2 时刻子线程申请不到锁 mutex，所以放弃 cpu，进入休眠；
4. n+3 时刻，主线程释放 mutex，离开临界区，并唤醒阻塞在 mutex 的子线程，子线程申请到 mutex，进入临界区；
5. n+4 时刻，子线程离开临界区，释放 mutex。

可以看到，加锁之后，即使主线程在 value2 =count; 之前产生了调度，子线程由于获取不到 mutex，会进入休眠，只有主线程出了临界区，子线程才能获得 mutex，访问 value1 和 value2,就永远不会打印信息，就实现了我们预期的代码时序。

总结

实际项目中，可能程序的并发的情况可能会更加复杂，比如**多个 cpu 上运行的任务之间，cpu 运行的任务和中断之间，中断和中断之间**，都有可能并发。

有些调度的概率虽然很小，但是不代表不发生，而且由于资源同步互斥导致的问题，很难复现，纵观 Linux 内核代码，所有的临界资源都会对应锁。

多阅读 Linux 内核源码，学向大神学习，与大神神交。

正所谓代码读百遍，其义自见！熟读代码千万行，不会编写也会抄！

关于内核和应用程序的同步互斥的知识点，可以查看一口君的其他文章。