操作系统八股文 之 线程管理

原创 C 陈同学在搬砖 2021-12-09 10:24

各位好 我是陈同学
一个放弃BAT offer的深圳老师
最近整理了自己的秋招笔记

我当初拿下BAToffer
就靠这份八股文
基本上涵盖了面试中80%的知识点
只要能记下这份八股文+做几个项目+刷一些题
就可以参加校招了
话不多说 正式开始分享

线程管理

Q1 为什么要引入线程

1.早期的计算机系统都只允许 一个任务独占系统资源，

一次只能执行一个程序 由于对程序并发执行的需求 引入了多进程

2.进程的引人可以解决多任务支持的问题，但是也产生了新的问题

问题一:每个进程分别分配资源开销比较大

问题二:进程频繁切换导致额外系统开销

问题三:进程间的通信实现复杂

3.考虑现实中的场景: 一个word程序如果采用多进程 一个进程负责界面交互

一个进程负责后台运算 会相当低效 (进程通信不好实现 进程频繁切换导致额外的系统开销)

一个同时要处理数以千记请求的网络数据库如果采用多进程 (对每个请求都创建一个进程去响应那服务器的资源肯定很快就耗尽了,而且进程切换消耗很大)

4.由此就演化出了在一个进程的内存空间上开辟多个"小进程"，利用这些小进程来实现多个任务的方法，这些小进程就是所谓的线程 这些线程在进程的内存空间内共享很多进程的资源

所以每个线程分配资源开销不会很大 线程的规模较小 故线程的切换开销也不会很大

线程之间共享进程的一部分地址空间 故线程之间的通信也不会很麻烦

从逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。

Q2 线程和进程之间的比较

线程和进程的联系

```
1.线程是进程里面的一个执行序列,
显然一个进程可以同时拥有多个执行序列,


2.每个独立的线程有一个程序运行的入口、
顺序执行序列和程序的出口。


 3.同一个进程中的多个线程之间可以并发执行.
 一个线程可以创建和撤销另一个线程;


4.但是线程不能够独立执行，必须依存在进程中


5.每个进程运行时,都会创建一个主线程,
也叫主控线程,
通过主控线程可以继续创建其他子线程
```

线程和进程的区别

线程和协程的比较

协程，是一种比线程更加轻量级的存在

正如一个进程可以拥有多个线程一样，
一个线程也可以拥有多个协程。

这样带来的好处就是性能得到了很大的提升，
不会像线程切换那样消耗资源

协程不是被操作系统内核所管理，
而完全是由程序所控制（也就是在用户态执行）。

见链接https://blog.csdn.net/vjhghjghj/article/details/105352264

Q3 线程的内存结构是什么样的?

- 线程间共享的资源:

在内核区:文件描述符表,每种信号的处理方式,当前工作目录

在用户区:堆区，数据区(bss段,Data段Test 段)

- 线程间独占的资源:

1.线程ID

每个线程都有自己的线程ID，这个ID在本进程中是唯一的。进程用此来标识线程。

2.寄存器组的值

由于线程间是并发运行的，每个线程有自己不同的运行线索，当从一个线 程切换到另一个线程上时， 必须将原有的线程的寄存器集合的状态保存，以便将来该线程在被重新切换到时能得以恢复。

3.错误返回码

由于同一个进程中有很多个线程在同时运行，可能某个线程进行系统调用后设置了errno值，而在该 线程还没有处理这个错误， 另外一个线程就在此时被调度器投入运行，这样错误值就有可能被修改。所以，不同的线程应该拥有自己的错误返回码变量。

4.线程的信号屏蔽码

由于每个线程所感兴趣的信号不同，所以线程的信号屏蔽码应该由线程自己管理。但所有的线程都 共享同样的信号处理器。

5.线程的优先级

由于线程需要像进程那样能够被调度，那么就必须要有可供调度使用的参数，这个参数就是线程的优先级。

6.线程的栈

Q4 线程的切换过程的什么样的

- （1）一般的进程切换分为两步：

[1] 切换页目录使用新的地址空间 [2] 切换内核栈和硬件上下文 对于Linux来讲，地址空间是线程和进程的最大区别，如果是线程切换的话，不需要做第一步，也就是切换页目录使用新的地址空间。但是切换内核栈和硬件上下文则是线程切换和进程切换都需要做的。

- （2）切换进程上下文：

进程上下文可以分为三个部分:

用户级上下文：正文、数据、用户堆栈以及共享存储区；

寄存器上下文：通用寄存器、程序寄存器(IP)、处理器状态寄存器(EFLAGS)、栈指针(ESP)；

系统级上下文：进程控制块task_struct、内存管理信息(mm_struct、vm_area_struct、pgd、pte)、内核栈。

系统中的每一个进程都有自己的上下文。一个正在使用处理器运行的进程称为当前进程(current)。

当前进程因时间片用完或者因等待某个事件而阻塞时，进程调度需要把处理器的使用权从当前进程交给另一个进程，这个过程叫做进程切换。此时，被调用进程成为当前进程。

在进程切换时系统要把当前进程的上下文保存在指定的内存区域（该进程的任务状态段TSS中），然后把下一个使用处理器运行的进程的上下文设置成当前进程的上下文。

当一个进程经过调度再次使用CPU运行时，系统要恢复该进程保存的上下文。所以，进程的切换也就是上下文切换。

- （3）线程切换：

Linux下的线程实质上是轻量级进程(light weighted process),线程生成时会生成对应的进程控制结构，只是该结构与父线程的进程控制结构共享了同一个进程内存空间。

同时新线程的进程控制结构将从父线程（进程）处复制得到同样的进程信息，如打开文件列表和信号阻塞掩码等。

创建线程比创建新进程成本低，因为新创建的线程使用的是当前进程的地址空间。

相对于在进程之间切换，在线程之间进行切换所需的时间更少，因为后者不包括地址空间之间的切换。

线程切换上下文切换的原理与此类似，只是线程在同一地址空间中，不需要MMU等切换，只需要切换必要的CPU寄存器，因此，线程切换比进程切换快的多。

Q5 线程的用户级和内核级是什么?

进程的实现只能由操作系统内核来实现，而不存在用户态实现的情况。

但是对于线程就不同了，线程的管理者可以是用户也可以是操作系统本身，

线程是进程内部的东西，当然存在由进程直接管理线程的可能性。

因此线程的实现就应该分为内核态线程实现和用户态线程实现。

内核态线程实现：

线程是进程的不同执行序列，也就是说线程是独立运行的基本单位，也是CPU调度的基本单位。

那么操作系统是如何实现管理线程的呢?

首先操作系统向管理进程一样，应该保持维护线程的所有资源，

将线程控制块存放在操作系统的内核空间中。

那么此时操作系统就同时掌管进程控制块和线程控制块。

操作系统管理线程的好处是：

1.用户编程简单；2.如果一个线程执行阻塞操作，操作系统可以从容的调度另外一个线程的执行。

内核线程的实现缺点是：

1.效率低，因为线程在内核态实现，每次线程切换都需要陷入到内核，由操作系统来调度，而有用户态切换到内核态是要话费很多时间的，

另外内核态实现会占用内核稀有的资源，因为操作系统要维护线程列表，操作系统所占内核空间一旦装载后就无法动态改变，

并且线程的数量远远大于进程的数量，随着线程数的增加内核将耗尽；

2.内核态的实现需要修改操作系统，这个是谁都不想要做的事情；

那么用户态是如何实现管理线程的呢?

用户态管理线程就是用户自己做线程的切换，自己管理线程的信息，操作系统无需知道线程的存在。

在用户态下进行线程的管理需要用户创建一个调度线程。

一个线程在执行完一段时间后主动把资源释放给其他线程使用，而在内核台下则无需如此，

因为操作系统可通过周期性的时钟中断把控制权夺过来，

在用户态实现情况下，执行系统的调度器也是线程，没有能力夺取控制权。

用户态实现有什么优点?

首先是灵活，因为操作系统不用知道线程的存在，

所以任何操作系统上都能应用；

其次，线程切换快，因为切换在用户态进行，无需陷入带内核态；

再次，不用修改操作系统实现容易。

用户态实现的缺点呢?

首先编程起来很诡异，由于在用户台下各个进程间需要相互合作才能正常运转。那么在编程时必须考虑什么情况下让出CPU，让其他的线程运行，而让出时机的选择对线程的效率和可靠性有很大影响，这个并不容易做到；

其次，用户态线程实现无法完全达到线程提出所要达到的目的：进程级多道编程；，如果在执行过程中一个线程受阻，它将无法将控制权交出来，这样整个进程都无法推进。操作系统随即把CPU控制权交给另外一个进程。这样，一个线程受阻造成整个进程受阻，我们期望的通过线程对进程实施分身的计划就失败了。这是用户态线程致命的缺点。

调度器激活：线程阻塞后，CPU控制权交给了操作系统，要激活受阻进程的线程，唯一的办法就是让操作系统在进程切换时先不切换，而是通知受阻的进程执行系统（即调用执行系统），并问其是否还有别的线程可以执行。如果有，将CPU控制权交给该受阻进程的执行系统线程，从而调度另一个可以执行的线程到CPU上。一个进程挂起后，操作系统并不立即切换到别的进程上，而是给该进程二次机会，让其继续执行。如果该进程只有一个线程，或者其所有线程都已经阻塞，则控制权将再次返回给操作系统。而现在，操作系统就会切换到其他线程了。

现在操作系统的线程实现模型：

鉴于用户态与内核态都存在缺陷，现代操作将两者结合起来。

用户态的执行负责进程内部线程在非阻塞时的切换；

内核态的操作系统负责阻塞线程的切换，即我们同时实现内核态和用户态线程管理。

每个内核态线程可以服务一个或者更多个用户态线程。

线程从用户态切换到内核态：

什么情况下会造成线程从用户态到内核态的切换呢?

首先，如果在程序运行过程中发生中断或者异常，

系统将自动切换到内核态来运行中断或异常处理机制。

此外，程序进行系统调用也会从用户态切换到内核态。

Q6 线程适用于哪些场合?

Q7 线程的属性有哪些?

- 介绍
- 工作过程

Q8 线程的接口函数有哪些?

线程标识函数pthread_self()

- 1.功能
- 2.使用原则

线程创建函数pthread_create()

- 功能

```
int pthread_create (pthread_ t *thread , canst pthread_attr_ t *attr, void * (*start_routine) (void *) , void *arg
第一个参数 传入参数  若线程创建成功
则该参数赋值指向线程标识符的指针
指向内存单元将存放线程id 若创建线程失败 则该参数未定义

第二个 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

返回值:若线程创建成功，则返回 0;
若线程创建失败，则返回出错编号,
```

- 使用原则

主线程回收子线程函数pthread_join()

- 功能

```
int pthread_join (pthread_t thread , void ** retval)

第一个参数为被 待的线程标识符,

第二个参数为一个用户定义的指针,
用来获取被等待线程的结束时候传进来的返回值

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,
函数返回时,被等待线程的资源被收回

代码中如果没有pthread_join主线程
会很快结束从而使整个进程结束,
从而使创建的线程没有机会开始执行就结束
```

子线程自我结束函数pthread_exit()

- 功能

```
pthread_exit 函数,

 一个线程的结束有两种途径:
 (1)函数已 结束,调用它的线程也就结束了;

(2)通过函数 pthread exit 来实现
它的函数原型为void pthread_exit (void *retval) ;
它的参数就是他要给主线程返回的值
```

- 使用原则

强制制止线程函数pthread_cancel()

- 使用原则

线程分离函数pthread_detach()

- 使用原则

线程的接口函数使用实例

```

```

```
#include<stdio.h>
#include<pthread.h>
#include<iostream>
#include<string>
using namespace std;
struct thread_info{
  unsigned long thread_id;
  string thread_name;
};
void* thread_func(void* argv )
{
 cout<<"now in thread...."<<endl;
 cout<<"main thread name:"<<(*(thread_info *)argv).thread_name<<endl;
   cout<<"main thread id:"<<(*(thread_info *)argv).thread_id<<endl;

 pthread_exit((void*)pthread_self());//结束时传出线程自己的id
}
int main()
{

  //下面结构体作为子线程参数
  thread_info main_thread_info;
  main_thread_info.thread_id=pthread_self();
  main_thread_info.thread_name="Kyle";
  //提前设置要创建线程的属性
  pthread_attr_t attr;
  int Ret=pthread_attr_init(&attr);
  if(Ret)
  {
    cout<<"thread   attrinit error!"<<endl;
  }
  Ret=pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
    if(Ret)
  {
    cout<<"thread set error!"<<endl;
  }
  //创建线程

  pthread_t tid;
  int Ret1= pthread_create(&tid,NULL,thread_func,(void *)&main_thread_info);
  if(Ret1)
  {
    cout<<"thread error:Return value="<<Ret1<<endl;
    return Ret1;
  }


  //回收线程(获取线程结束以后的状态量)
  void* Ret_val;
  int Ret2=pthread_join(tid,&Ret_val);
  if(Ret2)
  {
   cout<<"thread join error"<<endl;
   return Ret2;
  }
  cout<<"son thread id:"<<*((unsigned long *)Ret_val)<<endl;
  return 0;
}
```

Q8 线程进群是什么?

Q8 线程同步是什么?

Q9 线程同步有哪些措施?

有`互斥量，条件变量、信号量、读写锁、自旋锁`

Q10 介绍一下线程间的互斥量机制?

1.互斥量的理解

为了防止在售票窗口多人抢票时票务系统的出错

一种很容易想到的思路就是建立一个单独的售票间

房间是有锁的

这样顾客一个一个的来买票 每来一个就把售票间给锁上

买完以后就出去把锁打开
注意售票间外面的人是不需要排队的

每个人都会聚精会神的关注并等待锁打开，

谁先看到锁打开了就冲进去然后上锁

这里的售票间上的锁就是互斥量

Linux的pthreads mutex采用futex实现，

不必每次加锁 解锁都陷入系统调用

2.互斥量的接口函数

3.互斥量的使用实例

```
/********
 //模拟多个线程"抢票"过程
 //模拟用互斥量同步的情况下线程之间的抢占
 //设定总票数ticket_num为全局变量
 // 在每个线程中减1 表示买一张票
********/

#include<pthread.h>
#include<iostream>
#include<string>
#include<stdio.h>
#include<unistd.h>
using namespace std;
```

```
int ticket_nums=10;//票数作为全局变量
 pthread_mutex_t my_mutux;  //表示互斥量
 //也可以用下面这一句静态初始化互斥量的方式
 //如果用下面这一句来创建互斥量 就不用在主函数里面用pthread_mutex_init
 //pthread_mutex_t mutex_x= PTHREAD_MUTEX_ INITIALIZER ;
void* thread_func(void *ticket)
{

//每个线程内部轮询抢票15次
for(int i=0;i<15;i++)
{
    pthread_mutex_lock(&my_mutux);

  if((*(int *)ticket)>0)
    {

        (*(int *)ticket)--;
        cout<<"In thread"<<pthread_self()<<" Buy one ticket...."<<endl;

    }
    else
    {
         cout<<"In thread"<<pthread_self()<<"No Tickets Left...."<<endl;

    }
     pthread_mutex_unlock(&my_mutux);
     sleep(1);//注意这里的一秒钟休眠 让其他线程有机可乘
}
  //如果不想给主线程传递消息的话 pthread_exit也可以不加 函数结束 线程会自动结束
 pthread_exit(NULL);



}
 int main()
 {
 //并发多个线程  模拟多个线程抢票的过程
 int Ret;
 pthread_t tid[4];//表示线程id

pthread_mutex_init(&my_mutux,NULL);//表示互斥量初始化
 for(int i=0;i<4;i++)
 {
       //创建线程

        Ret=pthread_create(&tid[i],NULL,thread_func,&ticket_nums);
        if(Ret)
        {
            cout<<"creat thread error...."<<endl;
            return Ret;

        }
 }
// 主线程休眠5秒
//让子线程暂时脱离主线程的控制各自抢票
//如果此时主线程不休眠
//马上往下执行会立刻关闭子线程
//这样子线程就没有时间上演"抢票大战"
 sleep(10);
 for(int i=0;i<4;i++)
 {
       //销毁线程
        Ret=pthread_join(tid[i],NULL);
        if(Ret)
```

```
        {
            cout<<"destory thread error...."<<endl;
            return Ret;

        }
}



    return  0;

}
```

4.解决死锁问题

```
/***测试多线程死锁*****/

#include"MutexLock.h"
#include"MutexGuard.h"
#include<iostream>
#include<string>
#include<stdio.h>
#include<unistd.h>
#include<cassert>
using namespace std;
MutexLock my_mutex;

class DeadLockTest
{
    public:
        void printA()
    {
         MutexGuard  mutex_guardA( my_mutex);

              cout<<"now in A"<<endl;
              printB_nolock();//如果调用带锁版本的printB则会死锁  因为mutex非递归 不同一下子加两次锁

    }

         void printB_nolock() const
    {
              cout<<"now in B"<<endl;
    }
         void printB() const
    {

              MutexGuard  mutex_guardA( my_mutex);

              cout<<"now in B"<<endl;


    }
};
int main()
{
    DeadLockTest test;
    test.printA();
    return 0;


}
```

互斥量的RAII封装

RAII是Resource Acquisition Is Initialization （翻译成“资源获取即初始化”）的简称

Q11 介绍一下线程间的条件变量机制?

1.条件变量介绍

2.条件变量接口函数

3. 条件变量实例

```
/********
 //模拟多个线程"抢票"过程
 //模拟用互斥量和条件变量同步的情况下线程之间的抢占
 //设定总票数ticket_num为全局变量
 // 在每个线程中减1 表示买一张票
********/


#include<pthread.h>
#include<iostream>
#include<string>
#include<stdio.h>
#include<unistd.h>
using namespace std;
int ticket_num; //定义剩余票数 一开始定义为0 表示没有余票了
pthread_mutex_t mutex;//定义一个互斥锁
pthread_cond_t   con_var;//定义一个条件变量
void*  Buy_ticket(void* tmp)//买票线程
{
    pthread_mutex_lock(&mutex);
    while(!ticket_num)//这里应该用while 不能用if 避免虚假唤醒问题
    {
         cout<<"Thread"<<pthread_self()<<" start waiting......"<<endl;
        pthread_cond_wait(&con_var,&mutex);//如果发现没票了 就用条件变量让线程阻塞

    }
    ticket_num--;
    cout<<"Thread"<<pthread_self()<<" buy a ticket"<<endl;
    pthread_mutex_unlock(&mutex);

}




void*  Return_ticket(void* tmp)//买票线程
{
    pthread_mutex_lock(&mutex);
     ticket_num++;
       cout<<"Thread"<<pthread_self()<<" return  a ticket"<<endl;
    pthread_cond_signal(&con_var);//退一张票 通知等待队列里面的一个人可以买票了

    pthread_mutex_unlock(&mutex);

}

int main()
{
    pthread_t tid[4];//定义4个线程 其中3个线程买票 一个线程退票
    pthread_mutex_init(&mutex,NULL);//动态创建互斥锁
    pthread_cond_init(&con_var,NULL);//动态创建条件变量
    int Ret;
    //下面创建三个买票线程
    for(int i=0;i<3;i++)
    {
        Ret=pthread_create(&tid[i],NULL,Buy_ticket,NULL);
```

```
        if(Ret)
        {
            cout<<"thread creat fialed"<<endl;
             return Ret;
        }

    }

    //这里的暂停是为了让条件变量的效果显示的更明显一点
    // 让买票的的三个线程先在那等一会
    sleep(6);
    Ret=pthread_create(&tid[3],NULL,Return_ticket,NULL);
        if(Ret)
        {
            cout<<"thread creat fialed"<<endl;
            return Ret;
        }
    //这里的暂停是为了回收子线程
    //理论上此时只有一个线程买到票了 另外两个线程还在那等待
    //用暂停6秒的形式回收那两个还在等待的线程
    sleep(6);
    return 0;
}
```

```
```

Q12 介绍一下线程间的信号量机制?

1.信号量的介绍

2.信号量的接口函数

3.信号量的使用原则

4.信号量的实例

```
```

```
/********
 //模拟多个线程"抢票"过程
 //模拟用信号量同步的情况下线程之间的抢占
 // 在每个线程中减1 表示买一张票
********/

#include<pthread.h>
#include<iostream>
#include<string>
#include<stdio.h>
#include<unistd.h>
#include<semaphore.h>
using namespace std;
sem_t   ticket_num;//用信号量来存放票数

void* thread_func(void *thread_id)
{
    if(sem_wait(&ticket_num)==0)
    {
        cout<<"Thread "<<pthread_self()<<" buy a ticket"<<endl;
        usleep(100);
        sem_post(&ticket_num);
    }
    else
    {
         cout<<"Thread "<<pthread_self()<<" is waiting......"<<endl;
    }


  //如果不想给主线程传递消息的话 pthread_exit也可以不加 函数结束 线程会自动结束
 //pthread_exit(NULL);



}
 int main()
 {
 //并发多个线程   模拟多个线程抢票的过程
 int Ret;
 pthread_t tid[4];//表示线程id
sem_init(&ticket_num,0,2);//设定为局部信号量 票数设定为2

 for(int i=0;i<4;i++)
 {
       //创建线程
        Ret=pthread_create(&tid[i],NULL,thread_func,(void*)&i);
        if(Ret)
        {
            cout<<"creat thread error...."<<endl;
            return Ret;
        }
        usleep(10);

 }
sleep(5);
return  0;

 }
```

Q13 介绍一下线程间的读写锁机制?

1.读写锁介绍

现在咱们把场景切换到
自己的电脑上的文件读写场景，
假设有一个hello.txt文件
如果你用文本编辑器以写模式打开
则此时就不能再创建窗口打开它的
如果你用文本编辑器以只读模式打开
则此时你可以继续用只读模式打开它
但是不能用写写模式打开

读写锁就是这么回事 文绉绉一点去表达就是
1 ）当读写锁是写加锁状态时，在这个锁被解锁之前，
所有试图对这个锁加锁的线程都会被阻塞。

2）当读写锁在读加锁状态时，
所有试图以读模式对它进行加锁的线程都可以得到访问权，
但是以写模式对它进行加锁的线程将会被阻塞。
还要注意一点:假设一个线程读/写操作完了
有很多读/写进程会抢夺去访问 这个时候根据调度策略不同
如果此时先调度读线程则成为强读者同步 反之 则是强写者同步

2.读写锁接口函数

3.读写锁使用原则

4.读写锁实例

Q14 介绍一下线程间的自旋锁机制?

1.自旋锁的介绍

2.自旋锁的实现