###1.什么是线程

①在一个程序里的一个执行路线就叫做线程（thread）。更准确的定义是：线程是“一个进程内部的控制序列”

②一切进程至少都有一个执行线程

③线程在进程内部运行，本质是在进程地址空间内运行

④在Linux系统中，在CPU眼中，看到的PCB都要比传统的进程更加轻量化

透过进程虚拟地址空间，可以看到进程的大部分资源，将进程资源合理分配给每个执行流，就形成了线程执行流

\*\*\*\*拿自己话说：线程就是轻量级进程（LWP），线程tid不同，进程tgid相同

2.线程优点

①创建一个新线程的代价要比创建一个新进程小得多

②与进程之间的切换相比，线程之间的切换需要操作系统做的工作要少很多

③线程占用的资源要比进程少很多

④能充分利用多处理器的可并行数量

⑤在等待慢速I/O操作结束的同时，程序可执行其他的计算任务

⑥计算密集型应用，为了能在多处理器系统上运行，将计算分解到多个线程中实现

⑦I/O密集型应用，为了提高性能，将I/O操作重叠。线程可以同时等待不同的I/O操作。

3.线程缺点

①性能损失

②健壮性降低

③缺乏访问控制

④编程难度高

4.进程和线程的区别

①进程是资源竞争的基本单位，线程是程序执行的最小单位。

②在一个进程内的线程共享进程的 正文段和数据段，也就是定义一个函数，在各个线程中个都可以调用，如果定义一个全局变量， 在各线程中都可以访问到。

③除此之外，线程还共享以下资源和环境：文件描述符表、每种信号的处理方式、当前工作目录、用户id和组id。当然 ，还有 虚拟地址空间 和页表。

④线程拥有一部分自己的数据：如 线程ID、一组寄存器、栈空间、errno、信号屏蔽字、调度优先级。

###进程的多个线程共享 同一地址空间,因此Text Segment、Data Segment都是共享的,如果定义一个函数,在各线程中都可以调用,如果定义一个全局变量,在各线程中都可以访问到,除此之外,各线程还共享以下进程资源和环境:

文件描述符表

每种信号的处理方式(SIG\_ IGN、SIG\_ DFL或者自定义的信号处理函数)

当前工作目录

用户id和组id

**计算密集型程序适合C语言多线程，I/O密集型适合脚本语言开发的多线程。**

5.线程异常

①单个线程如果出现除零，野指针问题导致线程崩溃，进程也会随着崩溃

线程是进程的执行分支，线程出异常，就类似进程出异常，进而触发信号机制，终止进程，进程终止，该进程内的所有线程也就随即退出

6. 线程用途

①合理的使用多线程，能提高CPU密集型程序的执行效率

②合理的使用多线程，能提高IO密集型程序的用户体验（如生活中我们一边写代码一边下载开发工具，就是多线程运行的一种表现）

7.进程ID和线程ID

①在Linux中，目前的线程实现是Native POSIX Thread Libaray,简称NPTL。在这种实现下，线程又被称为轻量级进程(Light Weighted Process),每一个用户态的线程，在内核中都对应一个调度实体，也拥有自己的进程描

述符(task\_struct结构体)。

②没有线程之前，一个进程对应内核里的一个进程描述符，对应一个进程ID。但是引入线程概念之后，情况发生了变化，一个用户进程下管辖N个用户态线程，每个线程作为一个独立的调度实体在内核态都有自己的进程描述符，进程和内核的描述符一下子就变成了1：N关系，POSIX标准又要求进程内的所有线程调用，getpid函数时返回相同的进程ID，如何解决上述问题呢？

③Linux内核引入了线程组的概念。

多线程的进程，又被称为线程组，线程组内的每一个线程在内核之中都存在一个进程描述符（task\_struct）与之对应。进程描述符结构体中的pid，表面上看对应的是进程ID，其实不然，它对应的是线程ID;进程描述符中的tgid，含义是Thread Group ID,该值对应的是用户层面的进程ID。

④线程组内的第一个线程，在用户态被称为主线程(main thread),在内核中被称为group leader，内核在创建第一个线程时，会将线程组的ID的值设置成第一个线程的线程ID，group\_leader指针则指向自身，既主线程的进程描述符。所以线程组内存在一个线程ID等于进程ID，而该线程即为线程组的主线程。

8.

8.1线程创建

8.2 如果需要只终止某个线程而不终止整个进程,可以有三种方法:

①从线程函数return。这种方法对主线程不适用,从main函数return相当于调用exit。

②线程可以调用pthread\_ exit终止自己。

③一个线程可以调用pthread\_ cancel终止同一进程中的另一个线程。

*\*\*\*pthread\_exit*函数。功能：线程终止

原型：void pthread\_exit(void \*value\_ptr);

参数：value\_ptr:value\_ptr不要指向一个局部变量。

返回值：无返回值，跟进程一样，线程结束的时候无法返回到它的调用者（自身）

####需要注意,pthread\_exit或者return返回的指针所指向的内存单元必须是全局的或者是用malloc分配的,不能在线程函数的栈上分配,因为当其它线程得到这个返回指针时线程函数已经退出了。

*\*\*\*pthread\_cancel*函数：功能：取消一个执行中的线程

原型：int pthread\_cancel(pthread\_t thread);

参数：thread:线程ID

返回值：成功返回0；失败返回错误码

8.3线程等待 为什么需要线程等待？

原因：已经退出的线程，其空间没有被释放，仍然在进程的地址空间内。创建新的线程不会复用刚才退出线程的地址空间。

原型：int pthread\_join(pthread\_t thread, void \*\*value\_ptr);（joinable属性时调用，detach不可调用）

功能：等待线程结束

参数:thread:线程ID

value\_ptr:它指向一个指针，后者指向线程的返回值

返回值：成功返回0；失败返回错误码

说明：调用该函数的线程将挂起等待,直到id为thread的线程终止。thread线程以不同的方法终止,通过pthread\_join得到的终止状态是不同的，总结如下:

1. 如果thread线程通过return返回,value\_ ptr所指向的单元里存放的是thread线程函数的返回值。

2. 如果thread线程被别的线程调用pthread\_ cancel异常终掉,value\_ ptr所指向的单元里存放的是常数PTHREAD\_CANCELED。

3. 如果thread线程是自己调用pthread\_exit终止的,value\_ptr所指向的单元存放的是传给pthread\_exit的参数。

4. 如果对thread线程的终止状态不感兴趣，可以传NULL.

8.4 线程分离

默认情况下，新创建的线程是joinable的，线程退出后，需要对其进行pthread\_join操作，否则无法释放资源，从而造成系统泄漏。

如果不关心线程的返回值，join是一种负担，这个时候，我们可以告诉系统，当线程退出时，自动释放线程资源。可以是线程组内其他线程对目标线程进行分离，也可以是线程自己分离:

int pthread\_detach(pthread\_t thread);

pthread\_detach(pthread\_self());

joinable和分离是冲突的，一个线程不能既是joinable又是分离的。

8.5 Linux线程互斥

临界资源：多线程执行流共享的资源就叫做临界资源。

临界区：每个线程内部，访问临界自娱的代码，就叫做临界区

互斥：任何时刻，互斥保证有且只有一个执行流进入临界区，访问临界资源，通常对临界资源起保护作用

原子性（后面讨论如何实现）：不会被任何调度机制打断的操作，该操作只有两态，要么完成，要么未完成。

8.6互斥量mutex

大部分情况，线程使用的数据都是局部变量，变量的地址空间在线程栈空间内，这种情况，变量归属单个线程，其他线程无法获得这种变量。

但有时候，很多变量都需要在线程间共享，这样的变量称为共享变量，可以通过数据的共享，完成线程之间的交互。

多个线程并发的操作共享变量，会带来一些问题。

##要解决以上问题，需要做到三点：

①代码必须要有互斥行为：当代码进入临界区执行时，不允许其他线程进入该临界区。

②如果多个线程同时要求执行临界区的代码，并且临界区没有线程在执行，那么只能允许一个线程进入该临界区。

③如果线程不在临界区中执行，那么该线程不能阻止其他线程进入临界区。

要做到这三点，本质上就是需要一把锁。Linux上提供的这把锁叫互斥量。

8.7 互斥量的接口

8.7.1初始化互斥量

①静态分配：pthread\_mutex\_t mutex = PTHREAD\_1 MUTEX\_INITIALIZER

②动态分配：int pthread\_mutex\_init(pthread\_mutex\_t \*restrict mutex, const pthread\_mutexattr\_t \*restrict attr);

参数：mutex：要初始化的互斥量 attr：NULL

8.7.2 销毁互斥量

销毁互斥量需要注意：

①使用PTHREAD\_ MUTEX\_ INITIALIZER初始化的互斥量不需要销毁

②不要销毁一个已经加锁的互斥量

③已经销毁的互斥量，要确保后面不会有线程再尝试加锁

int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex)

8.7.3互斥量加锁和解锁

int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);

int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

返回值:成功返回0,失败返回错误号。

调用pthread\_ lock 时，可能会遇到以下情况:

①互斥量处于未锁状态，该函数会将互斥量锁定，同时返回成功

②发起函数调用时，其他线程已经锁定互斥量，或者存在其他线程同时申请互斥量，但没有竞争到互斥量，那么pthread\_ lock调用会陷入阻塞(执行流被挂起)，等待互斥量解锁。

8.8

###常见不可重入的情况

①调用了malloc/free函数，因为malloc函数是用全局链表来管理堆的

②调用了标准I/O库函数，标准I/O库的很多实现都以不可重入的方式使用全局数据结构。

③可重入函数体内使用了静态的数据结构

###常见可重入的情况

①不使用全局变量或静态变量

②不使用用malloc或者new开辟出的空间

③不调用不可重入函数

④不返回静态或全局数据，所有数据都有函数的调用者提供

⑤使用本地数据，或者通过制作全局数据的本地拷贝来保护全局数据

9.死锁

一组进程个各个进程占有不会被释放的资源的，互相申请被其他进程所占用不会释放的资源而处于的一种永久等待状态。

###死锁四个必要条件

互斥条件：一个资源每次只能被一个执行流使用

请求与保持条件：一个执行流因请求资源而阻塞时，对已获得的资源保持不放

不剥夺条件:一个执行流已获得的资源，在末使用完之前，不能强行剥夺

循环等待条件:若干执行流之间形成一种头尾相接的循环等待资源的关系

###避免死锁

破坏死锁的四个必要条件

加锁顺序一致

避免锁未释放的场景

资源一次性分配

###避免死锁算法

死锁检测算法(了解)

银行家算法（了解）

10. Linux线程同步

###条件变量

当一个线程互斥地访问某个变量时，它可能发现在其它线程改变状态之前，它什么也做不了。

例如：一个线程访问队列时，发现队列为空，它只能等待，只到其它线程将一个节点添加到队列中。这种情况就需要用到条件变量。

####同步概念与竞态条件

同步：在保证数据安全的前提下，让线程能够按照某种特定的顺序访问临界资源，从而有效避免饥饿问题，叫做同步。

竞态条件：因为时序问题，而导致程序异常，我们称之为竞态条件。

####条件变量函数

①初始化：int pthread\_cond\_init(pthread\_cond\_t \*restrict cond,const pthread\_condattr\_t \*restrict attr);

参数：cond：要初始化的条件变量 attr：NULL

②销毁：int pthread\_cond\_destroy

③等待条件满足：int pthread\_cond\_wait(pthread\_cond\_t \*restrict cond, pthread\_mutex\_t \*restrict mutex);

④唤醒等待：pthread\_cond\_signal(pthread\_cond\_t\* cond);

Pthread\_cond\_broadcast(pthread\_cond\_t\* cond);

11. 生产者消费者模型

###为何要使用生产者消费者模型 生产者消费者模式就是通过一个容器来解决生产者和消费者的强耦合问题。生产者和消费者彼此之间不直接通讯，而通过阻塞队列来进行通讯，所以生产者生产完数据之后不用等待消费者处理，直接扔给阻塞队列，消费者不找生产者要数据，而是直接从阻塞队列里取，阻塞队列就相当于一个缓冲区，平衡了生产者和消费者的处理能力。这个阻塞队列就是用来给生产者和消费者解耦的。

12. POSIX信号量

POSIX信号量和SystemV信号量作用相同，都是用于同步操作，达到无冲突的访问共享资源目的。 但POSIX可以用于线程间同步。

①初始化信号量

#include <semaphore.h>

int sem\_init(sem\_t \*sem, int pshared, unsigned int value);

参数：

pshared:0表示线程间共享，非零表示进程间共享

value：信号量初始值

②销毁信号量

int sem\_destroy(sem\_t \*sem);

③等待信号量

功能：等待信号量，会将信号量的值减1

int sem\_wait(sem\_t \*sem);

④

功能：发布信号量

表示资源使用完毕，可以归还资源了。将信号量值加1。

int sem\_post(sem\_t \*sem);

13. 读写锁

在编写多线程的时候，有一种情况是十分常见的。那就是，有些公共数据修改的机会比较少。相比较改写，它们读的机会反而高的多。通常而言，在读的过程中，往往伴随着查找的操作，中间耗时很长。给这种代码段加锁，会极大地降低我们程序的效率。那么有没有一种方法，可以专门处理这种多读少写的情况呢？ 有，那就是读写锁。

①初始化

Int pthread\_rwlock\_init(pthread\_rwlock\_t \*restrict, const pthread\_rwlockattr\_t\* restrict attr);

②销毁

Int pthread\_rwlock\_destroy(pthread\_1 rwlock\_t \*rwlock);

③加锁和解锁

int pthread\_rwlock\_rdlock(pthread\_rwlock\_t \*rwlock);

int pthread\_rwlock\_wrlock(pthread\_rwlock\_t \*rwlock);

int pthread\_rwlock\_unlock(pthread\_rwlock\_t \*rwlock);

14.线程池

15.单例模式

16.设计模式