题目 01：Linux 下线程的基本概念

线程的基本概念

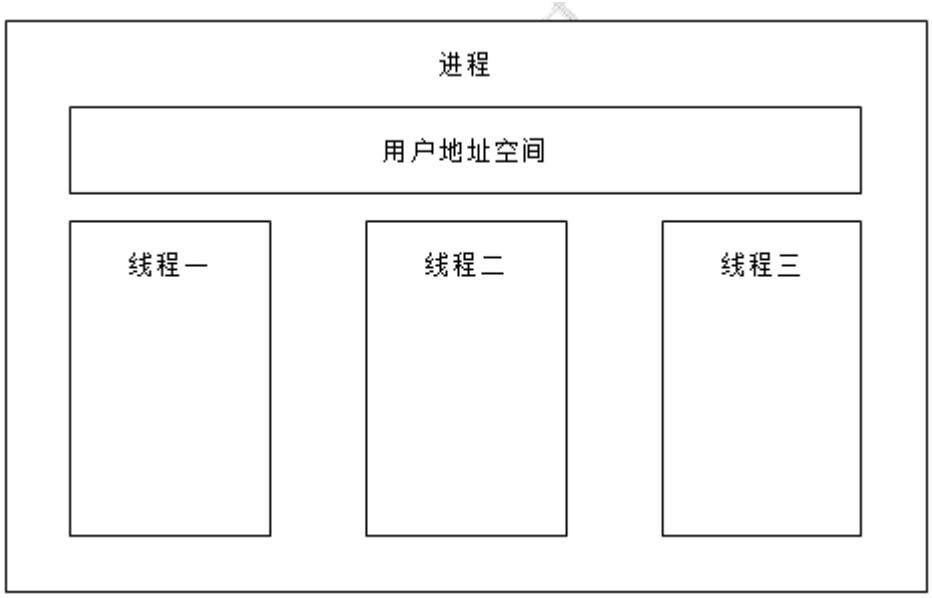
线程是操作系统能够进行调度运算的最小单位，它被包含在进程之中，是进程中的实际运作单位。一条线程指的是进程中一个单一顺序的控制流，一个进程中可以并发多个线程，每条线程并行执行不同的任务。

linux 操作系统使用符合 POSIX 线程作为系统标准线程，该 POSIX 线程标准定义了一整套操作线程的 API。

线程概述

线程是一个进程内的基本调度单位，也可以称为轻量级进程。线程是在共享内存空间中并发的多道执行路径，它们共享一个进程的资源，如文件描述和信号处理。因此，大大减少了上下文切换的开销。

同进程一样，线程也将相关的变量值放在线程控制表内。一个进程可以有多个线程，也就是有多个线程控制表及堆栈寄存器，但却共享一个用户地址空间。要注意的是，由于线程共享了进程的资源和地址空间,因此,任何线程对系统资源的操作都会给其他线程带来影响，因此，多线程中的同步就是非常重要的问题了。在多线程系统中，进程与进程的关系如表所示。



线程分类

线程按照其调度者可以分为用户级线程和核心级线程两种。

用户级线程

用户级线程主要解决的是上下文切换的问题,它的调度算法和调度过程全部由用户自行选择决定，在运行时不需要特定的内核支持。在这里，操作系统往往会提供一个用户空间的

线程库，该线程库提供了线程的创建、调度、撤销等功能，而内核仍然仅对进程进行管理。如果一个进程中的某一个线程调用了一个阻塞的系统调用，那么该进程包括该进程中的其他所有线程也同时被阻塞。这种用户级线程的主要缺点是在一个进程中的多个线程的调度中无法发挥多处理器的优势。

核心级线程

这种线程允许不同进程中的线程按照同一相对优先调度方法进行调度，这样就可以发挥多处理器的并发优势。现在大多数系统都采用用户级线程与核心级线程并存的方法。一个用户级线程可以对应一个或几个核心级线程，也就是“一对一”或“多对一”模型。这样既可满足多处理机系统的需要，也可以最大限度地减少调度开销。

Linux 线程技术的发展

在 Linux 中，线程技术也经过了一代代的发展过程。

在 Linux2.2 内核中，并不存在真正意义上的线程。当时 Linux 中常用的线程 pthread 实际上是通过进程来模拟的，也就是说 Linux 中的线程也是通过 fork 创建的“轻”进程，并且线程的个数也很有限，最多只能有 4096 个进程/线程同时运行。

Linux2.4 内核消除了这个线程个数的限制，并且允许在系统运行中动态地调整进程数上限。当时采用的是 LinuxThread 线程库，它对应的线程模型是“一对一”线程模型，也就是一个用户级线程对应一个内核线程，而线程之间的管理在内核外函数库中实现。这种线程模型得到了广泛应用。但是，LinuxThread 也由于 Linux 内核的限制以及实现难度等原因，并不是完全与 POSIX 兼容。另外，它的进程 ID、信号处理、线程总数、同步等各方面都还有诸多的问题。

为了解决以上问题，在 Linux2.6 内核中，进程调度通过重新编写，删除了以前版本中效率不高的算法。内核线程框架也被重新编写，开始使用 NPTL（Native POSIX Thread Library）线程库。这个线程库有以下几点设计目标：POSIX 兼容性、多处理器结构的应用、低启动开销、低链接开销、与 LinuxThreads 应用的二进制兼容性、软硬件的可扩展能力、与 C++集成等。这一切都使得 Linux2.6 内核的线程机制更加完备，能够更好地完成其设计目标。与 LinuxThreads 不同，NPTL 没有使用管理线程，核心线程的管理直接放在核内进行，这也带了性能的优化。由于 NPTL 仍然采用 1：1 的线程模型，NPTL 仍然不是 POSIX 完全兼容的， 但就性能而言相对 LinuxThreads 已经有很大程度上的改进。

多线程基本使用

线程基本操作

线程创建和退出

（1）函数说明

创建线程实际上就是确定调用该线程函数的入口点，这里通常使用的函数是 pthread_create。在线程创建以后，就开始运行相关的线程函数，在该函数运行完之后，该线程也就退出了，这也是线程退出一种方法。另一种退出线程的方法是使用函数 pthread_exit，这是线程的主动行为。这里要注意的是，在使用线程函数时，不能随意使用 exit 退出函数进行出错处理，由于 exit 的作用是使调用进程终止，往往一个进程包含多个线程，因此，在使用 exit 之后，该进程中的所有线程都终止了。因此，在线程中就可以使用 pthread_exit 来代替进程中的 exit。

由于一个进程中的多个线程是共享数据段的，因此通常在线程退出之后，退出线程所占

用的资源并不会随着线程的终止而得到释放。正如进程之间可以用 wait()系统调用来同步终止并释放资源一样，线程之间也有类似机制，那就是 pthread_join()函数。pthread_join 可以用于将当前线程挂起，等待线程的结束。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，被等待线程的资源就被收回。

（2）函数格式

pthread_create 函数语法要点

所需头文件	#include <pthread.h>
函数原型	int pthread_create ((pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg))
函数传入值	thread：线程标识符
	attr：线程属性设置（具体设定在 9.2.2 会进行讲解）
	start_routine：线程函数的起始地址
	arg：传递给 start_routine 的参数
函数返回值	成功：0
	出错： 1

pthread_exit 函数语法要点

所需头文件	#include <pthread.h>
函数原型	void pthread_exit(void *retval)
函数传入值	Retval：pthread_exit()调用者线程的返回值，可由其他函数如 pthread_join 来检索获取

pthread_join 函数语法要点

所需头文件	#include <pthread.h>
函数原型	int pthread_join ((pthread_t th, void **thread_return))
函数传入值	th：等待线程的标识符
	thread_return：用户定义的指针，用来存储被等待线程的返回值（不为 NULL 时）
函数返回值	成功：0
	出错： 1

（3）函数使用

以下实例中创建了两个线程，其中第一个线程是在程序运行到中途时调用 pthread_exit 函数退出，第二个线程正常运行退出。在主线程中收集这两个线程的退出信息，并释放资源。从这个实例中可以看出，这两个线程是并发运行的。

```c
#include <pthread.h>

/*线程一*/
void thread1(void) {
    int i=0;
    for(i=0; i<6; i++) {
        printf("This is a pthread1.\n");
        if(i==2)
            pthread_exit(0);
        sleep(1);
    }
}

/*线程二*/
void thread2(void) {
    int i;
    for(i=0; i<3; i++)
        printf("This is a pthread2.\n");
    pthread_exit(0);
}

int main(void) {
    pthread_t id1,id2;
    int i,ret;
    /*创建线程一*/
    ret=pthread_create(&id1,NULL,(void *) thread1,NULL);
    if(ret!=0) {
        printf ("Create pthread error!\n");
        exit (1);
    }
    /*创建线程二*/
    ret=pthread_create(&id2,NULL,(void *) thread2,NULL);
    if(ret!=0) {
        printf ("Create pthread error!\n");
        exit (1);
    }
    /*等待线程结束*/
    pthread_join(id1,NULL);
    pthread_join(id2,NULL);
    exit (0);
}
```

运行结果：

```
[root@RHEL74-SVR g00101]# ./thread_demo
This is a pthread1.
This is a pthread2.
This is a pthread2.
This is a pthread2.
This is a pthread1.
This is a pthread1.
```

线程同步的几种方式

线程同步方式有：互斥锁、条件变量、读写锁、信号量、自旋锁、屏障、原子操作等。

主要介绍多线程编程中三种线程同步机制：互斥锁、信号量、条件量。使用生产者消费者问题编程实践三种线程同步方式。

生产者、消费者问题：生产者线程生产物品，然后将物品放置在一个空缓冲区中供消费者线程消费。消费者线程从缓冲区中获得物品，然后释放缓冲区。当生产者线程生产物品时，如果没有空缓冲区可用，那么生产者线程必须等待消费者线程释放出一个空缓冲区。当消费者线程消费物品时，如果没有满的缓冲区，那么消费者线程将被阻塞，直到新的物品被生产出来。

一、互斥锁

互斥锁用来保证同一时间内只有一个线程在执行某段代码（临界区）。互斥锁可看作某种意义上的全局变量。在同一时刻只能有一个线程掌握某个互斥锁，拥有上锁状态的线程能够对共享资源进行操作。若其他线程希望上锁一个已经被上锁的互斥锁,则该线程就会挂起，直到上锁的线程释放掉互斥锁为止。互斥锁保证让每个线程对共享资源按顺序进行原子操作。

互斥锁主要包括下面的基本函数：

互斥锁初始化：pthread_mutex_init()

互斥锁上锁：pthread_mutex_lock()

互斥锁判断上锁：pthread_mutex_trylock()

互斥锁解锁：pthread_mutex_unlock()

消除互斥锁：pthread_mutex_destroy()

1、初始化互斥锁

静态分配：pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

动态分配：int pthread_mutex_init(pthread_mutex_t *mp, const pthread_mutexattr_t *mattr);

mp是互斥锁地址

mattr是属性，通常默认 null初始化互斥锁之前，必须将其所在的内存清零。如果互斥锁已初始化，则它会处于未锁定状态。互斥锁可以位于进程之间共享的内存中或者某个进程的专用内存中。

2、锁定互斥锁

int pthread_mutex_lock(pthread_mutex_t *mutex);

当pthread_mutex_lock() 返回时，该互斥锁已被锁定。调用线程是该互斥锁的属主。如果该互斥锁已被另一个线程锁定和拥有，则调用线程将阻塞，直到该互斥锁变为可用为止。

如果互斥锁类型为 PTHREAD_MUTEX_NORMAL，则不提供死锁检测。尝试重新锁定互斥锁会导致死锁。如果某个线程尝试解除锁定的互斥锁不是由该线程锁定或未锁定，则将产生不确定的行为。

如果互斥锁类型为 PTHREAD_MUTEX_ERRORCHECK，则会提供错误检查。如果某个线程尝试重新锁定的互斥锁已经由该线程锁定，则将返回错误。如果某个线程尝试解除锁定的互斥锁不是由该线程锁定或者未锁定，则将返回错误。

如果互斥锁类型为 PTHREAD_MUTEX_RECURSIVE，则该互斥锁会保留锁定计数这一概念。线程首次成功获取互斥锁时，锁定计数会设置为 1。线程每重新锁定该互斥锁一次，锁定计数就增加 1。线程每解除锁定该互斥锁一次，锁定计数就减小 1。 锁定计数达到 0 时，该互斥锁即可供其他线程获取。如果某个线程尝试解除锁定的互斥锁不是由该线程锁定或者未锁定，则将返回错误。

如果互斥锁类型是 PTHREAD_MUTEX_DEFAULT，则尝试以递归方式锁定该互斥锁将产生不确定的行为。对于不是由调用线程锁定的互斥锁，如果尝试解除对它的锁定，则会产生不确定的行为。如果尝试解除锁定尚未锁定的互斥锁，则会产生不确定的行为。

返回值：pthread_mutex_lock() 在成功完成之后会返回零。其他任何返回值都表示出现了错误。如果出现以下任一情况，该函数将失败并返回对应的值。

EAGAIN：由于已超出了互斥锁递归锁定的最大次数，因此无法获取该互斥锁。

EDEADLK：当前线程已经拥有互斥锁。

3、解除锁定互斥锁

int pthread_mutex_unlock(pthread_mutex_t *mutex);

函数说明：pthread_mutex_unlock() 可释放 mutex 引用的互斥锁对象。互斥锁的释放方式取决于互斥锁的类型属性。如果调用 pthread_mutex_unlock() 时有多个线程被 mutex 对象阻塞，则互斥锁变为可用时调度策略可确定获取该互斥锁的线程。对于 PTHREAD_MUTEX_RECURSIVE 类型的互斥锁，当计数达到零并且调用线程不再对该互斥锁进行任何锁定时，该互斥锁将变为可用。

返回值：pthread_mutex_unlock() 在成功完成之后会返回零。其他任何返回值都表示出现了错误。如果出现以下情况，该函数将失败并返回对应的值。

EPERM :当前线程不拥有互斥锁。

4、销毁互斥锁

int pthread_mutex_destroy(pthread_mutex_t *mp);

注意，没有释放用来存储互斥锁的空间。

返回值：pthread_mutex_destroy() 在成功完成之后会返回零。其他任何返回值都表示出现了错误。如果出现以下任一情况，该函数将失败并返回对应的值。

EINVAL: mp 指定的值不会引用已初始化的互斥锁对象。

5、程序实例

本实例使用互斥锁实现生产者、消费者的经典例子，生产者、消费者公用一个缓冲区，缓冲区存放一条消息。

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/time.h>

static char buff[50];
int msg_flag=0;
pthread_mutex_t mutex;

void consumeItem(char *buff) {
    printf("This is a consumer item\n");
}
void produceItem(char *buff) {
    printf("This is a produce item\n");
}
void *consumer(void *param) {
    while (1) {
        pthread_mutex_lock(&mutex);
        if (msg_flag > 0) {
            msg_flag--;
            consumeItem(buff);
        }
        pthread_mutex_unlock(&mutex);
        sleep(1);
    }
    return NULL;
}
```

```c
void *producer(void *param) {
    while (1) {
        pthread_mutex_lock(&mutex);
        if (msg_flag ==0 ) {
            msg_flag++;
            produceItem(buff);
        }
        pthread_mutex_unlock(&mutex);
        sleep(1);
    }
    return NULL;
}

int main() {
    pthread_t tid_c, tid_p;
    void *retval;
    pthread_mutex_init(&mutex, NULL);
    pthread_create(&tid_p, NULL, producer, NULL);
    pthread_create(&tid_c, NULL, consumer, NULL);

    pthread_join(tid_p, &retval);
    pthread_join(tid_c, &retval);

    return 0;
}
```

```
[root@RHEL74-SVR g00101]# ./mutex_demo
This is a produce item
This is a consumer item
This is a produce item
This is a consumer item
This is a produce item
This is a consumer item
This is a produce item
This is a consumer item
This is a produce item
```

二、信号量

与进程一样，线程也可以使用信号量来通信。线程使用信号量同步线程的步骤如下：

1、信号量初始化

```
int sem_init (sem_t *sem , int pshared, unsigned int value);
```

对sem指定的信号量进行初始化，设置好共享选项(linux只支持为0，即表示它是当前进程的局部信号量)，然后给它一个初始值VALUE。

2、等待信号量

```
int sem_wait(sem_t *sem);
```

给信号量减1，然后等待直到信号量的值大于0。

3、释放信号量

```
int sem_post(sem_t *sem);
```

信号量值加1。并通知其他等待线程。

4、销毁信号量

```
int sem_destroy(sem_t *sem);
```

5、程序实例

本实例使用条件变量实现生产者、消费者的经典例子，生产者、消费者共用一个缓冲区，缓冲区存放多条消息。使用信号量使用如下：

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

static char buff[50];
pthread_mutex_t mutex;
pthread_mutex_t cond_mutex;
pthread_cond_t cond;
sem_t msg_cnt;      //缓冲区消息数
sem_t idle_cnt;   //缓冲区空闲数

void consumeItem(char *buff) {
    printf("This is a consumer item\n");
}
void produceItem(char *buff) {
    printf("This is a produce item\n");
}
void *consumer(void *param) {
    while (1) {
        sem_wait(&msg_cnt);
        pthread_mutex_lock(&mutex);
        consumeItem(buff);
        pthread_mutex_unlock(&mutex);
        sem_post(&idle_cnt);
    }
    return NULL;
}
```

```c
void *producer(void *param) {
    while (1) {
        sem_wait(&idle_cnt);
        pthread_mutex_lock(&mutex);
        produceItem(buff);
        pthread_mutex_unlock(&mutex);
        sem_post(&msg_cnt);

    }
    return NULL;
}

int main() {
    pthread_t tid_c, tid_p;
    void *retval;
    pthread_mutex_init(&mutex, NULL);
    pthread_mutex_init(&cond_mutex, NULL);

    pthread_cond_t cond;
    pthread_cond_init(&cond, NULL);

    sem_init(&msg_cnt, 0, 0);        //初始缓冲区没有消息
    sem_init(&idle_cnt, 0, 10);  //初始缓冲区能放10条消息

    pthread_create(&tid_p, NULL, producer, NULL);
    pthread_create(&tid_c, NULL, consumer, NULL);


    pthread_join(tid_p, &retval);
    pthread_join(tid_c, &retval);

    return 0;
}
```

```
[root@RHEL74-SVR g00101]# ./sem_demo
This is a produce item
This is a produce item
This is a produce item
This is a produce item
This is a produce item
This is a produce item
This is a produce item
This is a produce item
This is a produce item
This is a produce item
This is a consumer item
This is a consumer item
This is a consumer item
This is a consumer item
This is a consumer item
This is a consumer item
This is a consumer item
This is a consumer item
This is a consumer item
This is a consumer item
```

三、条件变量(cond)

与互斥锁不同，条件变量是用来等待而不是用来上锁的。条件变量用来自动阻塞一个线程，直到某特殊情况发生为止。通常条件变量和互斥锁同时使用。条件变量分 为两部分：条件和变量。条件本身是由互斥量保护的。线程在改变条件状态前先要锁住互斥

量。条件变量使我们可以睡眠等待某种条件出现。条件变量是利用线程间共享的全局 变量进行同步的一种机制，主要包括两个动作：一个线程等待"条件变量的条件成立"而挂起；另一个线程使"条件成立"（给出条件成立信号）。条件的检测是在 互斥锁的保护下进行的。如果一个条件为假，一个线程自动阻塞，并释放等待状态改变的互斥锁。如果另一个线程改变了条件，它发信号给关联的条件变量，唤醒一个或多个等待它的线程，重新获得互斥锁，重新评价条件。如果两进程共享可读写的内存，条件变量可以被用来实现这两进程间的线程同步。

条件变量用来阻塞线程等待某个事件的发生，并且当等待的事件发生时，阻塞线程会被通知。互斥锁的缺点是只有两种状态：锁定和非锁定。而条件变量通过允许线程阻塞和等待另一个线程发送信号的方法弥补了互斥锁的不足，常和互斥锁一起使用。使用时，条件变量被用来阻塞一个线程，当条件不满足时，线程往往解开相应的互斥锁并等待条件发生变化。一旦其它的某个线程改变了条件变量，它将通知相应的条件变量唤醒一个或多个正被此条件变量阻塞的线程。这些线程将重新锁定互斥锁并重新测试条件是否满足。一般说来，条件变量被用来进行线承间的同步

1、初始化条件变量

静态初始化:pthread_cond_t cond = PTHREAD_COND_INITIALIER;

动态初始化：ead_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);

2、等待条件成立

释放锁,同时阻塞等待条件变量为真才行。timewait()设置等待时间,仍未signal,返回ETIMEOUT(加锁保证只有一个线程wait)

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

int thread_cond_timewait(pthread_cond_t *cond,pthread_mutex *mutex,const timespec *abstime);

3、激活条件变量

int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_broadcast(pthread_cond_t *cond); 解除所有线程的阻塞

4、清除条件变量

无线程等待,否则返回EBUSY

int pthread_cond_destroy(pthread_cond_t *cond);

5、程序实例

本实例使用条件变量实现生产者、消费者的经典例子，生产者、消费者公用一个缓冲区，缓冲区存放多条消息，有多个消费者。如果都使用互斥锁，那么多个消费者线程都要不断的去查看缓冲区是否有消息，有就取走。如果我们用条件变量，多个消费者线程都可以放心的“睡觉”，缓冲区有消息，消费者会收到通知，进而收取消息就行。

```c
include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/time.h>

static char buff[50];
pthread_mutex_t mutex;
pthread_mutex_t cond_mutex;
pthread_cond_t cond;

void consumeItem(char *buff) {
    printf("This is a consumer item\n");
}
void produceItem(char *buff) {
    printf("This is a produce item\n");
}
void *consumer(void *param) {
    int t = *(int *)param;
    while (1) {
        pthread_cond_wait(&cond, &cond_mutex);
        pthread_mutex_lock(&mutex);
        printf("consumer thread %d: ", t);
        consumeItem(buff);
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
```

```c
void *producer(void *param) {
    while (1) {
        pthread_mutex_lock(&mutex);
        produceItem(buff);
        pthread_mutex_unlock(&mutex);
        pthread_cond_signal(&cond);
        sleep(1);
    }
    return NULL;
}

int main(int argc, char **argv) {
    pthread_t tid_c1, tid_p, tid_c2, tid_c3;
    void *retval;
    pthread_mutex_init(&mutex, NULL);
    pthread_mutex_init(&cond_mutex, NULL);

    pthread_cond_t cond;
    pthread_cond_init(&cond, NULL);


    int p[3] = {1, 2, 3}; //线程编号
    pthread_create(&tid_p, NULL, producer, NULL);
    pthread_create(&tid_c1, NULL, consumer, &p[0]);
    pthread_create(&tid_c2, NULL, consumer, &p[1]);
    pthread_create(&tid_c3, NULL, consumer, &p[2]);


    pthread_join(tid_p, &retval);
    pthread_join(tid_c1, &retval);
    pthread_join(tid_c2, &retval);
    pthread_join(tid_c3, &retval);

    return 0;
}
```

```
[root@RHEL74-SVR 00101]# ./cond_demo
This is a produce item
consumer thread 1: This is a consumer item
This is a produce item
consumer thread 1: This is a consumer item
This is a produce item
consumer thread 2: This is a consumer item
This is a produce item
consumer thread 3: This is a consumer item
This is a produce item
consumer thread 1: This is a consumer item
This is a produce item
consumer thread 2: This is a consumer item
This is a produce item
consumer thread 3: This is a consumer item
```

线程等待与唤醒

暂时省略。

线程和进程异同点

1).二者的相同点

无论是进程还是线程，都是用来实现多任务并发的技术手段；

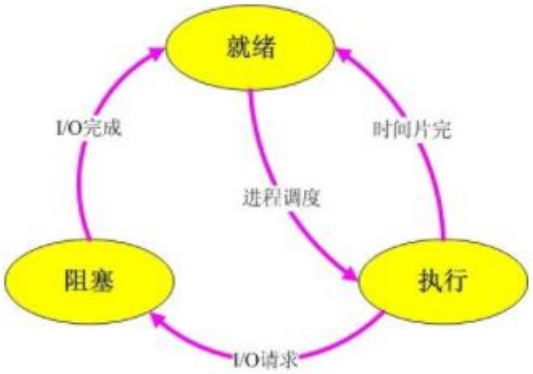
二者都可以独立调度，因此在多任务环境下，功能上并无差异；

二者都具有各自的实体，是系统独立管理的对象个体。所以在系统层面，都可以通过技术手段实现二者的控制；

二者所具有的状态都非常相似；

在多任务程序中，子进程(子线程)的调度一般与父进程(父线程)平等竞争。

其实在Linux内核2.4版以前，线程的实现和管理方式就是完全按照进程方式实现的。在2.6版内核以后才有了单独的线程实现。



进程状态图

线程状态图

2).实现方式的差异

进程是资源分配的基本单位，线程是调度的基本单位。

进程的个体间是完全独立的，而线程间是彼此依存的。多进程环境中，任何一个进程的终止，不会影响到其他进程。而多线程环境中，父线程终止，全部子线程被迫终止(没有了资源)。而任何一个子线程终止一般不会影响其他线程，除非子线程执行了exit()系统调用。任何一个子线程执行exit()，全部线程同时灭亡。

多线程程序中至少有一个主线程，而这个主线程其实就是有main函数的进程。它是整个程序的进程，所有线程都是它的子线程。我们通常把具有多线程的主进程称之为主线程。

从系统实现角度讲，进程的实现是调用fork系统调用：pid_t fork(void);

线程的实现是调用clone系统调用：int clone(int (*fn)(void *), void *child_stack, int flags, void *arg, .../* pid_t *ptid, struct user_desc *tls, pid_t *ctid */);

其中，fork()是将父进程的全部资源复制给了子进程。而线程的clone只是复制了一小部分必要的资源。在调用clone时可以通过参数控制要复制的对象。可以说，fork实现

的是clone的加强完整版。

实际中，编写多进程程序时采用fork创建子进程实体。而创建线程时并不采用clone系统调用，而是采用线程库函数。常用线程库有Linux-Native线程库和POSIX线程库。其中应用最为广泛的是POSIX线程库。因此读者在多线程程序中看到的是pthread_create而非clone。

我们知道，库是建立在操作系统层面上的功能集合，因而它的功能都是操作系统提供的。由此可知，线程库的内部很可能实现了clone的调用。不管是进程还是线程的实体，都是操作系统上运行的实体。

vfork()也是一个系统调用，用来创建一个新的进程。它创建的进程并不复制父进程的资源空间，而是共享，也就说实际上vfork实现的是一个接近线程的实体，只是以进程方式来管理它。并且，vfork()的子进程与父进程的运行时间是确定的：子进程“结束”后父进程才运行。

3).多任务程序设计模式的区别

由于进程间是独立的，所以在设计多进程程序时，需要做到资源独立管理时就有了天然优势，而线程就显得麻烦多了。比如多任务的TCP程序的服务端，父进程执行accept()一个客户端连接请求之后会返回一个新建立的连接的描述符DES，此时如果fork()一个子进程，将DES带入到子进程空间去处理该连接的请求，父进程继续accept等待别的客户端连接请求，这样设计非常简练，而且父进程可以用同一变量(val)保存accept()的返回值，因为子进程会复制val到自己空间，父进程再覆盖此前的值不影响子进程工作。但是如果换成多线程，父线程就不能复用一个变量val多次执行accept()了。因为子线程没有复制val的存储空间，而是使用父线程的，如果子线程在读取val时父线程接受了另一个客户端请求覆盖了该值，则子线程无法继续处理上一次的连接任务了。改进的办法是子线程立马复制val的值在自己的栈区，但父线程必须保证子线程复制动作完成之后再执行新的accept()。但这执行起来并不简单，因为子线程与父线程的调度是独立的，父线程无法知道子线程何时复制完毕。这又得发生线程间通信，子线程复制完成后主动通知父线程。这样一来父线程的处理动作必然不能连贯，比起多进程环境，父线程显得效率有所下降。

关于资源不独立，看似是个缺点，但在有的情况下就成了优点。多进程环境间完全独立，要实现通信的话就得采用进程间的通信方式，它们通常都是耗时间的。而线程则不用任何手段数据就是共享的。当然多个子线程在同时执行写入操作时需要实现互斥，否则数据就写“脏”了。

4).实体间(进程间，线程间，进线程间)通信方式的不同

进程间的通信方式有这样几种：A.共享内存 B.消息队列 C.信号量 D.有名管道 E.无名管道 F.信号 G.文件 H.socket

线程间的通信方式上述进程间的方式都可沿用，且还有自己独特的几种：A.互斥量 B.自旋锁 C.条件变量 D.读写锁 E.线程信号 G.全局变量

值得注意的是，线程间通信用的信号不能采用进程间的信号，因为信号是基于进程

为单位的，而线程是共属于同一进程空间的。故而要采用线程信号。

综上，进程间通信手段有8种。线程间通信手段有13种。

进程间采用的通信方式要么需要切换内核上下文，要么要与外设访问(有名管道，文件)。所以速度会比较慢。而线程采用自己特有的通信方式的话，基本都在自己的进程空间内完成，不存在切换，所以通信速度会较快。也就是说，进程间与线程间分别采用的通信方式，除了种类的区别外，还有速度上的区别。

5).控制方式的异同

进程与线程的身份标示ID管理方式不一样，进程的ID为pid_t类型，实际为一个int型的变量(也就是说是有限的)：

```
/usr/include/unistd.h:260:typedef __pid_t    pid_t;
/usr/include/bits/types.h:126:# define __STD_TYPE    typedef
/usr/include/bits/types.h:142:__STD_TYPE  __PID_T_TYPE    __pid_t;
/usr/include/bits/typesizes.h:53:#define __PID_T_TYPE    __S32_TYPE
/usr/include/bits/types.h:100:#define    __S32_TYPE      int
```

在全系统中，进程ID是唯一标识，对于进程的管理都是通过PID来实现的。每创建一个进程，内核去中就会创建一个结构体来存储该进程的全部信息：

注：下述代码来自 Linux内核3.18.1

```
include/linux/sched.h:1235:struct task_struct {
  volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
  void *stack;
  ...
  pid_t pid;
  pid_t tgid;
  ...
};
```

每一个存储进程信息的节点也都保存着自己的PID。需要管理该进程时就通过这个ID来实现(比如发送信号)。当子进程结束要回收时(子进程调用exit()退出或代码执行完)，需要通过wait()系统调用来进行，未回收的消亡进程会成为僵尸进程，其进程实体已经不复存在，但会虚占PID资源，因此回收是有必要的。

线程的ID是一个long型变量：

```
/usr/include/bits/pthreadtypes.h:60:typedef unsigned long int pthread_t;
```

它的范围大得多，管理方式也不一样。线程ID一般在本进程空间内作用就可以了，当然系统在管理线程时也需要记录其信息。其方式是，在内核创建一个内核态线程与之对应，也就是说每一个用户创建的线程都有一个内核态线程对应。但这种对应关系不是

一对一，而是多对一的关系，也就是一个内核态线程可以对应着多个用户级线程。

对于线程而言，若要主动终止需要调用pthread_exit() ，主线程需要调用pthread_join()来回收(前提是该线程没有被detached。

6).资源管理方式的异同

进程本身是资源分配的基本单位，因而它的资源都是独立的，如果有多进程间的共享资源，就要用到进程间的通信方式了，比如共享内存。共享数据就放在共享内存去，大家都可以访问，为保证数据写入的安全，加上信号量一同使用。一般而言，共享内存都是和信号量一起使用。消息队列则不同，由于消息的收发是原子操作，因而自动实现了互斥，单独使用就是安全的。

线程间要使用共享资源不需要用共享内存，直接使用全局变量即可，或者malloc()动态申请内存。显得方便直接。而且互斥使用的是同一进程空间内的互斥量，所以效率上也有优势。

实际中，为了使程序内资源充分规整，也都采用共享内存来存储核心数据。不管进程还是线程，都采用这种方式。原因之一就是，共享内存是脱离进程的资源，如果进程发生意外终止的话，共享内存可以独立存在不会被回收(是否回收由用户编程实现)。进程的空间在进程崩溃的那一刻也被系统回收了。虽然有coredump机制，但也只能是有限的弥补。共享内存在进程down之后还完整保存，这样可以拿来分析程序的故障原因。同时，运行的宝贵数据没有丢失，程序重启之后还能继续处理之前未完成的任务，这也是采用共享内存的又一大好处。

总结之，进程间的通信方式都是脱离于进程本身存在的，是全系统都可见的。这样一来，进程的单点故障并不会损毁数据，当然这不一定全是优点。比如，进程崩溃前对信号量加锁，崩溃后重启，然后再次进入运行状态，此时直接进行加锁，可能造成死锁，程序再也无法继续运转。

7).个体间辈分关系的迥异

进程的备份关系森严，在父进程没有结束前，所有的子进程都尊从父子关系，也就是说A创建了B，则A与B是父子关系，B又创建了C，则B与C也是父子关系，A与C构成爷孙关系，也就是说C是A的孙子进程。在系统上使用pstree命令打印进程树，可以清晰看到备份关系。

多线程间的关系没有那么严格，不管是父线程还是子线程创建了新的线程，都是共享父线程的资源，所以，都可以说是父线程的子线程，也就是只存在一个父线程，其余线程都是父线程的子线程。

8).进程池与线程池的技术实现差别

进程和线程的创建时需要时间的，并且系统所能承受的进程和线程数也是有上限的，这样一来，如果业务在运行中需要动态创建子进程或线程时，系统无法承受不能立即创建的话，必然影响业务。综上，聪明的程序员发明了一种新方法——池。

在程序启动时，就预先创建一些子进程或线程，这样在需要用时直接使唤。这就是老人口中的“多生孩子多种树”。程序才开始运行，没有那么多的服务请求，必然大量

的进程或线程空闲，这时候一般让他们“冬眠”，这样不耗资源，要不然一大堆孩子的口食也是个负担啊。对于进程和线程而言，方式是不一样的。另外，当你有了任务，要分配给那些孩子的时候，手段也不一样。下面就分别来解说。

进程池

首先创建了一批进程，就得管理，也就是你得分开保存进程ID，可以用数组，也可用链表。建议用数组，这样可以实现常数内找到某个线程，而且既然做了进程池，就预先估计好了生产多少进程合适，一般也不会再动态延展。就算要动态延展，也能预估范围，提前做一个足够大的数组。不为别的，就是为了快速响应。本来错进程池的目的也是为了效率。

接下来就要让闲置进程冬眠了，可以让他们pause()挂起，也可用信号量挂起，还可以用IPC阻塞，方法很多，分析各自优缺点根据实际情况采用就是了。

然后是分配任务了，当你有任务的时候就要让他干活了。唤醒了进程，让它从哪儿开始干呢？肯定得用到进程间通信了，比如信号唤醒它，然后让它在预先指定的地方去读取任务，可以用函数指针来实现，要让它干什么，就在约定的地方设置代码段指针。这也只是告诉了它怎么干，还没说干什么(数据条件)，再通过共享内存把要处理的数据设置好，这也子进程就知道怎么做了。干完之后再来一次进程间通信然后自己继续冬眠，父进程就知道孩子干完了，收割成果。

最后结束时回收子进程，向各进程发送信号唤醒，改变激活状态让其主动结束，然后逐个wait()就可以了。

线程池

线程池的思想与上述类似，只是它更为轻量级，所以调度起来不用等待额外的资源。要让线程阻塞，用条件变量就是了，需要干活的时候父线程改变条件，子线程就被激活。线程间通信方式就不用赘述了，不用繁琐的通信就能达成，比起进程间效率要高一些。线程干完之后自己再改变条件，这样父线程也就知道该收割成果了。整个程序结束时，逐个改变条件并改变激活状态让子线程结束，最后逐个回收即可。