进程（Process）和线程（Thread）都是操作系统中的基本概念，它们之间有一些优劣和差异。

# 进程

## 概述

进程是程序执行时的一个实例，是系统进行资源分配的基本单位。

另一种表述：进程是一个具有一定独立功能的程序在一个数据集合上的一次**动态执行过程**。

所有与该进程有关的资源，都被记录在**进程控制块(PCB)**中。以表示该进程拥有这些资源或正在使用它们。

另外，进程也是抢占处理机的调度单位，它拥有一个完整的虚拟地址空间。当进程发生调度时，不同的进程拥有不同的虚拟地址空间，而同一进程内的不同线程共享同一地址空间。

进程ID：每个Linux进程都一定有一个唯一的数字标识符，称为进程ID。进程ID总是一非负整数。

### 组成

一个进程应该包括：

1. 程序的代码；
2. 程序处理的数据；
3. 程序计数器中的值，指示下一条将运行的指令；
4. 一组通用的寄存器的当前值，栈、堆；
5. 一组系统资源（如打开的文件）。

总之，进程包含了正在运行的一个程序的所有状态信息。

### 进程与程序



程序是存放在磁盘文件中的可执行文件。

进程是程序的执行实例。

**进程与程序的联系：**

1. 程序是产生进程的基础；
2. 程序的每次运行构成不同的进程；
3. 进程是程序功能的体现；
4. 通过多次执行，一个程序可以对应多个进程；通过调用关系，一个进程可以包括多个程序。

**进程与程序的区别：**

1. 进程是动态的，程序是静态的：程序是有序代码的集合，进程是程序的执行，进程有内核态和用户态；
2. 进程是暂时的，程序是永久的：进程是一个状态变化的过程，程序可以长久保存；
3. 进程与程序的组成不同：进程的组成包括程序、数据和进程控制块（即进程状态信息）。

### 进程与线程

进行和线程之间的差异可以从下面几个方面来阐述：

**调度：**在引入线程的操作系统中，**线程是调度和分配的基本单位，进程是资源拥有的基本单位**。把传统进程的两个属性分开，线程便能轻装运行，从而可 显著地提高系统的并发程度。在同一进程中，线程的切换不会引起进程的切换；在由一个进程中的线程切换到另一个进程中的线程时，才会引起进程的切换。

**并发性：**在引入线程的操作系统中，不仅进程之间可以并发执行，而且在一个进程中的多个线程之间亦可并发执行，因而使操作系统具有更好的并发性，从而能更有效地使用系统资源和提高系统吞吐量。

**拥有资源：**不论是传统的操作系统，还是设有线程的操作系统，进程都是拥有资源的一个独立单位，它可以拥有自己的资源。一般地说，线程自己不拥有系统资源（只有一些必不可少的资源，但它可以访问其隶属进程的资源。



**系统开销：**由于在创建或撤消进程时，系统都要为之分配或回收资源，因此，操作系统所付出的开销将显著地大于在创建或撤消线程时的开销。**进程切换的开销也远大于线程切换的开销**。

**通信：**进程间通信IPC，线程间可以直接读写进程数据段（如全局变量）来进行通信——需要进程同步和互斥手段的辅助，以保证数据的一致性，因此共享简单。但是线程的数据同步要比进程略复杂。

从上面的分析可以看到，似乎线程有很多优势，比如，数据共享效率高，可应对并发操作，有效利用等待时间等等，但是多线程的编程比多进程要复杂，同时，多进程的可靠性较好，因为进程间不会相互影响。实际情况还是需要自己分析拿捏的。但是一般来说，实际应用中常常采用“进程+线程”结合的方式，而不是非此即彼，因为它们两者没有绝对的好与不好，而是适合于不同场景。

### 线程与协程

协程，英文Coroutines，是一种基于线程之上，但又比线程更加轻量级的存在，这种由程序员自己写程序来管理的轻量级线程叫做用户空间线程（用户态，不需要内核态，**减少线程上下文切换**），具有**对内核来说不可见的特性（线程是需要内核态）**。

协程是为了**解决线程上下文切换（即用户态和内核态切换）的开销**，将线程切换全部都放在用户态。

在传统的J2EE系统中都是基于每个请求占用一个线程去完成完整的业务逻辑（包括事务）。所以系统的吞吐能力取决于每个线程的操作耗时。如果遇到很耗时的I/O行为，则整个系统的吞吐立刻下降，因为这个时候线程一直处于阻塞状态，如果线程很多的时候，会存在很多线程处于空闲状态（等待该线程执行完才能执行），造成了资源应用不彻底。

最常见的例子就是JDBC（它是同步阻塞的），这也是为什么很多人都说数据库是瓶颈的原因。这里的耗时其实是让CPU一直在等待I/O返回，说白了线程根本没有利用CPU去做运算，而是处于空转状态。而另外过多的线程，也会带来更多的ContextSwitch开销。

对于上述问题，现阶段行业里的比较流行的解决方案之一就是单线程加上异步回调。其代表派是node.js以及Java里的新秀Vert.x。

而协程的目的就是当出现长时间的I/O操作时，通过让出目前的协程调度，执行下一个任务的方式，来消除ContextSwitch上的开销。

### 特点

动态性：可动态地创建、结束进程；

并发性：进程可以被独立调度并占用处理机运行（并发并行）；

独立性：不同进程的工作不相互影响；

制约性：因访问共享数据/资源或进程间同步而产生制约。

### 进程控制结构

**进程控制块：**操作系统管理控制进程运行所用的信息集合。操作系统用PCB来描述进程的基本情况以及运行变化的过程，PCB是进程存在的唯一标志。

**使用进程控制块：**

进程的创建：为该进程生成一个PCB；

进程的终止：回收它的PCB；

进程的组织管理：通过对PCB的组织管理来实现。

**PCB包含三大类信息：**

1. 进程标识信息：如本进程的标识，本进程的产生着标识（父进程标识），用户标识；
2. 处理机状态信息保存区：保存进程的运行现场信息：
3. 用户可见寄存器，用户程序可以使用的数据，地址等寄存器；
4. 控制和状态寄存器，如程序计数器（PC），程序状态字（PSW）；
5. 栈指针，过程调用/系统调用/中断处理和返回时需要用到它。
6. 进程控制信息：
7. 调度和状态信息，用于操作系统调度进程并占用处理机使用；
8. 进程间通信信息，为支持进程间的与通信相关的各种标识、信号、信件等，这些信息存在接收方的进程控制块中；
9. 存储管理信息，包含有指向本进程映像存储空间的数据结构；
10. 进程所用资源，说明由进程打开、使用的系统资源，如打开的文件等；
11. 有关数据结构连接信息，进程可以连接到一个进程队列中，或连接到相关的其他进程的PCB。

**PCB的组织方式：**

链表：同一状态的进程其PCB组成一链表，多个状态对应多个不同的链表。各状态的进程组成不同的链表：就绪链表、阻塞链表。

索引表：同一状态的进程引入一个index表（由index指向PCB），多个状态对应多个不同的index表。各状态的运行形成不同的索引表：就绪索引表表、阻塞索引表。



## 分类

这几种IPC的关系：



### Unix IPC

UnixIPC包括：管道、FIFO、信号

### System IPC

SystemV IPC包括：SystemV消息队列、SystemV信号量、SystemV共享内存

### Posix IPC

Posix IPC包括：Posix消息队列、Posix信号量、Posix共享内存区

### Linux IPC

## 进程状态

对于前面所述进程组成、控制块等属于进程的静态信息，这里介绍一下进程的动态信息。

### 进程的生命周期

#### 进程创建

引起进程创建的三个主要事件：

1. 系统初始化时；
2. 用户请求创建一个新进程；
3. 正在运行的进程执行了创建进程的系统调用。

#### 进程运行

内核选择一个就绪的进程，让它占用处理机并执行。

这里涉及两个问题：

1. 为何选择？
2. 如何选择？

#### 进程等待

在以下情况下，进程等待（阻塞）：

1. 请求并等待系统服务，无法马上完成；
2. 启动某种操作，无法马上完成；
3. 需要的数据没有到达。

进程只能自己阻塞自己，因为只有进程自身才能知道何时需要等待某种事件的发生。

#### 进程唤醒

进程唤醒的原因：

1. 被阻塞进程需要的资源可被满足；
2. 被阻塞进程等待的事件到达；
3. 将该进程的PCB插入到就绪队列。

进程只能被别的进程或操作系统唤醒。

#### 进程结束

在以下四种情况下，进程结束：

1. 正常退出（自愿的）
2. 错误退出（自愿的）
3. 致命错误（强制性的）
4. 被其他进程所杀（强制性的）

### 进程状态变化模型

进程的三种基本状态：进程在生命结束前处于仅处于三种基本状态之一，不同系统设置的进程状态数目不同。

运行状态（Running）：当一个进程正在处理机上运行时。

就绪状态（Ready）：一个进程获得了除处理机之外的一切所需资源，一旦得到处理机即可运行。

等待状态（又称阻塞状态Blocked）：一个进程正在等待某一事件而暂停运行时。如等待某资源，等待输入/输出完成。



进程其他的基本状态：

创建状态（New）：一个进程正在被创建，还没被转到就绪状态之前的状态。

结束状态（Exit）：一个进程正在从系统中消失时的状态，这是因为进程结束或由于其他原因所导致。

**状态变化图：**



可能的状态变化如下：

NULL->New：一个新进程被产生出来执行一个程序。

New->Ready：当进程被创建完成并初始化后，一切就绪准备运行时，变为就绪状态（很快）。

Ready->Running：处于就绪状态的进程被进程调度程序选中后，就分配到处理机上来运行。

Running->Exit：当进程表示它已经完成或者因出错，当前运行进程会由操作系统作结束处理。

Running->Ready：处于运行状态的进程在其运行过程中，由于分配给它的处理机时间片用完而让出处理机。

Running->Blocked：当进程请求某样东西且必须等待时。

Blocked->Ready：当进程要等待某事件到来时，它从阻塞状态变到就绪状态。

### 进程挂起

之所以引入进程挂起，是为了合理并充分利用系统资源。

进程在挂起状态时，意味着进程没有占用内存空间。处于挂起 状态的进程映像在磁盘上。



**挂起状态：**

阻塞挂起状态（Blocked-suspend）：进程在外存并等待某事件的出现；

就绪挂起状态（Ready-suspend）：进程在外存，但只要进入内存，即可运行。

**与挂起相关的状态转换：**

挂起（suspend）：把一个进程从内存转到外存，可能存在以下几种情况：

1. 阻塞到阻塞挂起：没有进程处于就绪状态或就绪进程要求更多内存资源时，会进行这种转换，以提交新进程或运行就绪进程；
2. 就绪到就绪挂起：当有高优先级阻塞（系统认为会很快就绪的）进程和低优先就绪进程时，系统会选择挂起低优先级就绪进程；
3. 运行到就绪挂起：对抢先式分时系统，当有高优先级阻塞挂起进程因事件出现而进入就绪挂起时，系统可能会把运行进程转到就绪挂起状态。

在外存时的状态转换：

1、阻塞挂起到就绪挂起：当有阻塞挂起进程因相关事件出现时，系统会把阻塞挂起进程转换为就绪挂起进程；

## 进程环境

### main函数和参数

#### main函数

格式：

int main(int argc,char \*\*argv)

初启函数：

认为：C程序从main开始执行

实际：真正启动点是系统提供的一个初启函数

位置：位于库文件中(crt0.o)

装配：ld在链接时候（加到main函数前面），形成可执行文件

作用：参数组织，调用main，中止进程

**实例：**

#include <stdio.h>

int main(int argc,char \*\*argv)

{

return 0;

}

#### 命令行参数

格式：

以空格分开的字符串

例如：$./a.out –f foo test

POSIX命令行参数语法约定：

选项、选项值（选项操作数）、操作数（命令操作数）

短横线开头，选项

多个选项，无选项值，可以放在一起

选项先于操作数

两个短横线终止所有选项

多个选项可任意顺序出现，可多次出现

扫描命令行中的选项：

选项如何提取：未知选项

选项值如何提取：选项如何对应选项值，缺少选项值

操作数如何获取：

报错：

扫描命令行中的选项：getopt函数

**实例：**

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

int main(int argc, char \*argv[])

{

int flags, opt;

int nsecs, tfnd;

int i;

nsecs = 0;

tfnd = 0;

flags = 0;

while ((opt = getopt(argc, argv, "nt:")) != -1) {

switch (opt) {

case 'n':

flags = 1;

break;

case 't':

nsecs = atoi(optarg);

tfnd = 1;

break;

default: /\* '?' \*/

fprintf(stderr, "Usage: %s [-t nsecs] [-n] name\n",

argv[0]);

exit(EXIT\_FAILURE);

}

}

printf("flags=%d; tfnd=%d; optind=%d\n", flags, tfnd, optind);

if (optind >= argc) {

fprintf(stderr, "Expected argument after options\n");

exit(EXIT\_FAILURE);

}

for(i=optind;i<argc;i++)

{

printf("name argument = %s\n", argv[i]);

}

/\* Other code omitted \*/

exit(EXIT\_SUCCESS);

}

#### 环境变量参数

argv为程序的某一次特定运行参数

不常改变或系统共享信息

标准环境变量名：env和export命令设置查看

环境表

访问环境

##### 环境表



实例：

#include <stdio.h>

extern char \*\*environ;

int main(int argc,char \*\*argv)

{

char \*\* p = NULL;

for(p=environ;NULL!=\*p;p++)

{

printf("%s\n",\*p);

}

return 0;

}

##### 环境变量操作

进程中环境变量操作：

#include<stdio.h>

char \*getenv(const char \*name);

返回：指向与name关联的value的指针，若未找到则返回NULL

int putenv(char \*str);

int setenv(const char \*name, const char \*value, int rewrite);

//第3个参数表示是否覆盖原有环境变量

int unsetenv(const char \*name);

返回：成功返回0，出错返回非0

实例：

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

extern char \*\*environ;

void showenv()

{

int i = 0;

char \*str;

while((str = environ[i]) != NULL)

{

printf("%s\n",str);

i++;

}

}

int main(void)

{

showenv();

printf("----------------------------\n");

printf("PATH:%s\n",getenv("PATH"));

setenv("PATH",".:/bin:/usr/bin",1);

printf("PATH:%s\n",getenv("PATH"));

putenv("NAME=jiekexueyuan");

printf("NAME:%s\n",getenv("NAME"));

return 0;

}

### 终止进程

进程终止流程：



注：\_exit通知内核，把进程占据的资源释放，而exit是用户态的终止函数，通知终止处理函数，IO清理函数及内核。

如果直接调用\_exit则标准IO未清除，一些资源没有在终止处理函数中释放，则有可能存在问题。

#### 正常终止

正常终止的几种方式：

1、从main返回

2、调用exit（标准库提供）

3、调用\_exit或\_Exit（内核提供）

4、最后一个线程从其启动例程返回

5、最后一个线程调用pthread\_exit

##### 终止函数

**exit和\_exit区别：**

flush I/O：exit是先刷新缓冲区（调用atexit函数刷新缓冲区），然后结束进程，而\_exit是不刷新缓冲区（内核直接调用，没有经过atexit），直接从进程退出

实例：

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <errno.h>

int main(int argc,char \*argv[])

{

if(argc < 3)

{

fprintf(stderr,"usage : %s file return | exit | \_exit\n",argv[0]);

exit(1);

}

FILE \*fp = fopen(argv[1],"w");

char \*str = "helloworld";

fprintf(fp,"%s",str);

if(!strcmp(argv[2],"return")) //命令行输入结束方式

{

return 0; //结束方式直接return，会刷新输出helloword

}else if(!strcmp(argv[2],"exit"))

{

exit(0); //exit退出，会刷新输出helloword

}else if(!strcmp(argv[2],"\_exit"))

{

\_exit(0); //\_exit退出，不会刷新输出helloword

}

else

{

printf("process error\n");

}

return 0;

}

##### 出口状态

有两种方式：exit(0)和return 0;

通常返回值0/1采用宏定义SUCCESS或FILED。

##### 终止前的清理

实例：

#include <stdio.h>

#include <stdlib.h>

void fun(void)

{

//printf("clean something!!!\n");

}

int main(int argc,char \*\*argv)

{

printf("hello world");

atexit(fun);//注册函数

atexit(fun);//注册函数

exit(EXIT\_SUCCESS);

return 0;

}

#### 异常终止

异常终止进程的方法：

1、调用abort（与信号有关）

2、接受到一个信号并终止（和信号有关）

3、最后一个线程对取消请求作出响应

##### abort函数

实例：

#include <stdio.h>

#include <stdlib.h>

void fun(void)

{

printf("clear someting!!!\n");

}

int main(int argc,char \*\*argv)

{

printf("hello world");

atexit(fun);

abort();

return 0;

}

#include <stdio.h>

#include <stdlib.h>

void fun(void)

{

printf("clean something!!!\n");

}

int main(int argc,char \*\*argv)

{

printf("hello world");

atexit(fun);//注册函数

atexit(fun);//注册函数

\_exit(EXIT\_SUCCESS);

return 0;

}

##### core文件

##### 信号

### 进程存储空间

#### 存储结构



#### 进程地址空间





使用命令size查看各段内存占据的大小。

#### 内存分配/释放

堆上分配

边界对齐

分配大小合适

内存泄露

实例：

#include <stdio.h>

#include <stdlib.h>

#define MAX 100

int main(int argc,char \*\*argv)

{

char \*buf = NULL;

//申请100字节内存，堆

buf = malloc(MAX);

if ( NULL == buf )

{

perror("malloc");

return -1;

}

while(1)

{

//memset 将内存清0

memset(buf,'\0',MAX);

fgets(buf,MAX,stdin);

if(0==strncmp("exit",buf,4))

{

exit(0);

}

printf("%s\n",buf);

}

free(buf);

buf = NULL;

}

### 非局部转移

非局部跳转应注意的问题：

1. 自动，寄存器和易失变量

寄存器中的变量在非局部跳转时可以恢复原始值。

1. 自动变量的潜在问题

#### setjmp

int setjmp(jmp\_buf env);

返回：直接调用返回0，若从longjump调用返回，则返回非0值

注：类似goto语句，但是goto语句只能在函数内部跳转，非局部跳转可以在栈区不同函数之间跳转。

#### longjmp

void longjmp(jmp\_buf env, int val);

实例：

#include <stdio.h>

#include <setjmp.h>

static jmp\_buf jmpbuf;

int a = 0;

void fun(int a,int b)

{

printf("in fun a is %d, b is %d\n",a,b);

longjmp(jmpbuf,3);

}

int main(int argc,char \*\*argv)

{

int b = 1;

if( 3 == setjmp(jmpbuf) )

{

printf("a is %d\nb is %d\n",a,b);

exit(0);

}

a = 2;

b = 3;

fun(a,b);

return 0;

}

**实例：**

#include <stdio.h>

#include <stdlib.h>

#include <setjmp.h>

jmp\_buf jmp;

int g\_v = 1;

void fun()

{

longjmp(jmp,1);

}

int main(void)

{

static int s\_v = 1;

auto int a\_v = 1;

register r\_v = 1;

volatile v\_v = 1;

int \*h\_v = (int\*)malloc(sizeof(int));

\*h\_v = 1;

printf("g\_v=%d,s\_v=%d,a\_v=%d,r\_v=%d,v\_v=%d,h\_v=%d\n",

g\_v,s\_v,a\_v,r\_v,v\_v,\*h\_v);

if(setjmp(jmp) == 1)

{

printf("g\_v=%d,s\_v=%d,a\_v=%d,r\_v=%d,v\_v=%d,h\_v=%d\n",

g\_v,s\_v,a\_v,r\_v,v\_v,\*h\_v);

exit(1);

}

g\_v = 10;

s\_v = 10;

a\_v = 10;

r\_v = 10;

v\_v = 10;

\*h\_v = 10;

printf("g\_v=%d,s\_v=%d,a\_v=%d,r\_v=%d,v\_v=%d,h\_v=%d\n",

g\_v,s\_v,a\_v,r\_v,v\_v,\*h\_v);

fun();

return 0;

}

### 进程的资源

#### 资源限制

Linux中可用的资源限制：

RLIMIT\_AS：进程可用存储区大小

RLIMIT\_CORE：core文件最大字节数

RLIMIT\_CPU：CPU时间最大值

RLIMIT\_DATA：数据段最大长度

RLIMIT\_FSIZE：可创建文件的最大长度

RLIMIT\_LOCKS：文件锁的最大数

RLIMIT\_MEMLOCK：使用mlock能否在存储器中锁定的最长字节数

RLIMIT\_NOFILE：能打开的最大文件数

RLIMIT\_NPROC：每个用户ID可拥有的最大子进程数

RLIMIT\_RSS：最大驻内存集的字节长度

RLIMIT\_STACH：栈的最大长度

在Linux系统下可以通过修改/etc/security/limit.conf配置文件实现系统软硬资源的限制修改（一般实际应用中这里是注释掉的，程序都是自己设定自己的资源限制，也可以在这里修改系统的资源限制）。

#### 查看资源

##### 虚拟空间的最大字节

##### core文件的最大字节

##### CPU时间最大值

##### 数据段空间的最大字节

##### 创建的最大文件的字节数

##### 在物理内存中上锁的最大虚拟存储字节数

##### 最多能够打开的文件个数

##### 创建的子进程个数

##### 最大物理内存页数

##### 栈的最大字节数

##### 不同资源的个数

实例：

#include <stdio.h>

#include <stdlib.h>

#include <sys/time.h>

#include <sys/resource.h>

int main(int argc,char \*\*argv)

{

struct rlimit limit;

getrlimit(RLIMIT\_CORE,&limit);

printf("soft limit core is %d\n",limit.rlim\_cur);

printf("hard limit core is %d\n",limit.rlim\_max);

limit.rlim\_cur = limit.rlim\_max;

setrlimit(RLIMIT\_CORE,&limit);

abort();

return 0;

}

#### 设置资源

进程资源限制：

#include<sys/resource.h>

int getlimit(int resource, struct rlimit \*rlptr);

int setlimit(int resource, const struct rlimit \*rlptr);

返回：成功返回0，出错返回非0

struct rlimit{

rlim\_t rlim\_cur;/\*soft limit,current limit\*/

rlim\_t rlim\_max;/\*hard limit,maxium limit\*/

}

资源限制的修改规则：

1. 任何一个进程可以降低或者提升其软资源限制、但必须小于等于其硬资源限制
2. 任何一个进程可以降低其硬件资源限制，但必须大于其软限制，普通用户不可逆此操作（软资源限制<硬资源限制）
3. 只有root用户可以提升硬件资源限制

Linux中进程资源的初始值由进程0建立，并被后续进程继承。

实例：

#include <sys/resource.h>

#include <stdio.h>

#include <errno.h>

#include <stdlib.h>

#include <string.h>

#define LIMIT(resource)\

query\_limit(#resource,resource)

void query\_limit(char \*name,int res)

{

struct rlimit limit;

if(getrlimit(res,&limit))

{

fprintf(stderr,"getrlimit error %s\n",strerror(errno));

exit(1);

}

printf("%15s ",name);

if(limit.rlim\_cur == RLIM\_INFINITY) //如果无限制

{

printf(" soft (INFINITY)");

}else

{

printf(" soft (%8ld)",limit.rlim\_cur);

}

if(limit.rlim\_max == RLIM\_INFINITY)

{

printf(" hard (INFINITY)");

}else

{

printf(" hard (%8ld)",limit.rlim\_max);

}

printf("\n");

}

int main(void)

{

LIMIT(RLIMIT\_AS);

LIMIT(RLIMIT\_CORE);

LIMIT(RLIMIT\_CPU);

LIMIT(RLIMIT\_DATA);

LIMIT(RLIMIT\_FSIZE);

LIMIT(RLIMIT\_LOCKS);

LIMIT(RLIMIT\_MEMLOCK);

LIMIT(RLIMIT\_NOFILE);

struct rlimit lt;

lt.rlim\_cur = 4;

lt.rlim\_max = 1024;

if(setrlimit(RLIMIT\_NOFILE,&lt))

{

fprintf(stderr,"setrlimit error %s\n",strerror(errno));

exit(1);

}

printf("----------------------------\n");

LIMIT(RLIMIT\_NOFILE);

FILE \*fp1 = fopen("a.txt","r");

if(fp1 == NULL)

{

fprintf(stderr,"file:%s\n",strerror(errno));

exit(1);

}else

{

printf("open sucessed\n");

}

FILE \*fp2 = fopen("a.txt","r");

if(fp2 == NULL)

{

fprintf(stderr,"file:%s\n",strerror(errno));

exit(1);

}else

{

printf("open sucessed\n");

}

return 0;

}

#### 统计资源

### 用户信息

#### 获取用户名

#### 用户数据库管理

#### 组数据库管理

**实例：**

#include <stdio.h>

#include <unistd.h>

int main(int argc,char \*\*argv)

{

printf("%s\n",getlogin());

return 0;

}

### 进程身份及调整

#### 身份凭证

#### 调整身份凭证

## 进程控制

### 常规操作

#### fork

原型：pid\_t fork(void)

功能：创建子进程

返回值：fork调用一次，返回两次，有三种不同的返回值

1. 父进程中，fork返回新进程的子进程PID
2. 子进程中，fork返回0
3. 如果出现错误，fork返回一个负值

注意：使用此函数创建的子进程，其数据空间、堆栈空间都会从父进程得到一个拷贝，而不是共享。

在Linux系统中，用户创建进程的唯一方法就是使用系统调用fork，其大致操作：

1. 分配表项，一个用户的进程项是有限的；
2. 创建子进程的进程标识号；
3. 复制父进程中的项目给子进程；
4. 与父进程相连的文件表和索引值加1，与子进程相连；
5. 内核为子进程创建用户级上下文；
6. 生成进程的动态部分。

#### vfork

除了不拷贝父进程的页表项外，vfork()系统调用和fork()的功能相同。子进程作为父进程的一个单独的进程在它的地址空间里运行，父进程被阻塞，直到子进程退出或执行exec()。子进程不能向地址空间写入。

在过去的3BSD时期，这个优化是很有意义的，那时并未使用写时拷贝页来实现fork()。现在由于在执行fork()时引入了写时拷贝页并且明确了子进程先执行，vfork()的好处就仅限于不拷贝父进程的页表项了。如果Linux将来fork()有了写时拷贝页表项，那么vfork()就彻底没用了。理想情况下，系统最好不要调用vfork()，内核也不用实现它。完全可以把vfrok()实现成一个普通的fork()——实际上，Linux2.2以前都是这么做的。

vfork与fork创建子进程的区别：

1. **fork子进程拷贝父进程的数据段，vfork子进程与父进程共享数据段**
2. **fork父子进程的执行次序不确定，vfork子进程先执行，然后执行父进程**

#### setsid

当进程是会话的领头进程时，setsid调用成功返回新的会话ID，失败并返回-1。调用sedsid函数的进程称为新的会话领头进程，并与其父进程的会话组和进程组脱离。由于会话对控制终端的独占性，进程同时与控制终端脱离。

setsid后子进程不受终端影响，终端退出，不影响子进程，即设置守护进程（我们在SecureCRT启动该进程时，它可以单独运行，退出控制终端后也不会停止）。

编写守护进程的一般步骤：

1. 在父进程中执行fork并exit退出；
2. 在子进程中调用setsid函数创建新的会话；
3. 在子进程中调用chdir函数，让根目录“/”成为子进程的工作目录；
4. 在子进程中调用umask函数，设置进程的umask为0；
5. 在子进程中关闭任何不需要的文件描述符。

#### exec

exec系统调用是一个函数蔟：

int execl(const char \*path,const char \*arg);

int execlp(const char \*file,const char \*arg);

int execle(const char \*path,const char \*arg, char \*const envp[]);

int execv(const char \*path,const char \*arg[]);

int execvp(const char \*file,const char \*arg[]);

exec函数最大的作用就是可以取代调用进程的内容，也是Linux可以执行新程序的关键。

**exec与fork区别：**

1. fork创建一个新的进程，产生一个新的PID；
2. exec启动一个新的程序，替换原来的进程，进程的PID不会改变。

当fork完后调用exec便会运行新的进程，这也是为何fork赋值出来的子进程执行之后与父进程不一样的根本原因。

#### exit

函数exit就是进程结束函数，就是一个进程结束的标识。

exit与\_exit函数区别：

1. exit定义在stdlib.h中，\_exit定义在unistd.h中；
2. 它们都是写入内存，但是\_exit只有在满足特点的条件下才能写入文件，如何要保存数据的完整性，一般使用exit函数。

#### getpid/getppid

pid\_t getpid(void) //获取本进程ID

pid\_t getppid(void) //获取父进程ID

#### wait

原型：pid\_t wait(int \*status)

参数：status

在wait中，这是一个指向一个整型数据的指针，用来存放子进程退出时的状态，后来又定义了一个WIFEXITED（status）的宏来完成这个工作，这里的status就是一个指针指向的函数。

功能：阻塞该进程，直到某个子进程退出（用于结束子进程）。

进程同步：通过wait在父进程中等待子进程的返回值。

僵尸进程：当一个进程调用exit之后，该进程并非马上消息还留下一些残余的信息。可以使用wait结束僵尸进程，进程一旦调用wait，就会立即阻塞自己，当分析到当前进程的子进程已经exit，便会收集这个子进程的信息，然后彻底销毁，如果没有找到这样的子进程，就会一直阻塞在这里，直到有一个出现。

#### sleep

函数sleep就是挂起进程指定的秒数，时间到了返回0。

#### kill

### 特殊操作

#### setuid

设置特权进程的函数：setsid、setgid。setsid设置用户ID位，setgid设置组ID位。

当一个程序一旦设置了这个标志以后，运行该程序的进程将拥有该程序所有者同样的权限。

setsid提升使用者的权限，普通用户可以执行该命令，使自己升级为root（比如chmod）。

#### setgid

与setsid类似，使得使用者在执行该文件时，都绑定了文件所有组的权限，单独setgid的文件非常少用，通常都是setsid的同时setgid。

#### setpgrp

#### setpgid

#### chdir

#### chroot

#### nice

### 创建进程

进程ID

父进程ID

进程组ID

用户表识

#### 写时拷贝

#### 获取进程标识符

#include<sys/types.h>

#include<unistd.h>

pid\_getpid(void); 返回：调用进程的进程ID

pid\_getppid(void); 返回：调用进程的父进程ID

uid\_t getuid(void); 返回：调用进程的实际用户ID

uid\_geteuid(void); 返回：调用进程的有效用户ID

uid\_getgid(void); 返回：调用进程的实际租ID

uid\_getegid(void); 返回：调用进程的有效组ID

#### 进程创建

#include<sys/types.h>

#include<unistd.h>

pid\_t fork(void);

返回：子进程中为0，父进程中为子进程ID，出错为-1

pid\_t vfork(void);

返回：子进程中为0，父进程中为子进程ID，出错为-1

子进程一般继承父进程：用户信息/权限目录信息信号信息环境表共享存储段资源限制。

子进程特有属性：父进程 id，锁(记录在 i 节点中 (struct flock 中有 pid))，运行时间，未绝信号等。

实例：

#include <stdio.h>

#include <fcntl.h>

#include <unistd.h>

#include <string.h>

#include <errno.h>

#include <stdlib.h>

#include <sys/wait.h>

int main(void)

{

int fd = open("a.txt",O\_RDWR);

if(fd < 0)

{

fprintf(stderr,"open error %s\n",strerror(errno));

exit(1);

}

pid\_t pid;

pid = fork();

if (pid < 0)

{

fprintf(stderr,"fork error %s\n",strerror(errno));

exit(1);

}else if (pid > 0) //parent process

{

if(lseek(fd,0L,2) < 0)

{

fprintf(stderr,"lseek error %s\n",strerror(errno));

exit(1);

}

close(fd);

}

else

{

sleep(3);

char \*str = "abcdefg";

write(fd,str,strlen(str));

}

close(fd);

wait(0);

return 0;

}

fork()：子进程拷贝父进程的数据段，代码段。

vfork()：子进程与父进程共享数据段。

fork()：父子进程的执行次序不确定(有内核调度程序决定)。

vfork ：保证子进程先运行，在调用exec 或exit 之前与父进程数据是共享的(在进程的数据空间中运行)，在它调用exec或exit之后父进程才可能被调度运行。

vfork()：保证子进程先运行，调用exec或exit 之后父进程才可能被调度运行。如果在调用这两个函数之前子进程依赖于父进程的进一步动作，则会导致死锁。

实例：

#include <unistd.h>

#include <string.h>

#include <errno.h>

#include <stdlib.h>

#include <stdio.h>

int main(void)

{

pid\_t pid;

pid = fork();

if(pid < 0)

{

fprintf(stderr,"fork error: %s\n",strerror(errno));

exit(1);

}else if(pid > 0)

{

sleep(3);

printf("parent pid = %d,ppid = %d,child pid = %d\n",getpid(),getppid(),pid);

}else

{

printf("child pid = %d,ppid = %d\n",getpid(),getppid());

}

printf("process finished\n");

return 0;

}

#### 僵死进程

##### 产生

当你运行一个程序时，它会产生一个父进程以及很多子进程。所有这些子进程都会消耗内核分配给它们的内存和CPU资源。

这些子进程完成执行后会发送一个Exit信号然后死掉。这个Exit信号需要被父进程所读取。父进程需要随后调用wait命令来读取子进程的退出状态，并将子进程从进程表中移除。

若父进程正确第读取了子进程的Exit信号，则子进程会从进程表中删掉。

但若父进程未能读取到子进程的Exit信号，则这个子进程虽然完成执行处于死亡的状态，但也不会从进程表中删掉。

##### 危害

父进程没有退出，子进程退出，但是父进程没有回收子进程的退出资源，此时子进程就是僵死进程。(由于此时期父进程没有退出，所以 init 进程不会去领养)。僵死进程则造成资源浪费。

由于僵尸进程并不做任何事情， 不会使用任何资源也不会影响其它进程， 因此存在僵尸进程也没什么坏处。 不过由于进程表中的退出状态以及其它一些进程信息也是存储在内存中的，因此存在太多僵尸进程有时也会是一些问题。

##### 查找

打开终端并输入下面命令：

ps aux | grep Z

会列出进程表中所有僵尸进程的详细内容。

##### 杀死/规避

正常情况下我们可以用SIGKILL信号来杀死进程，但是僵尸进程已经死了， 你不能杀死已经死掉的东西。因此你需要输入的命令应该是：

kill -s SIGCHLD pid

将这里的pid替换成父进程的进程 id，这样父进程就会删除所有以及完成并死掉的子进程了。

避免僵死进程：

**让子进程成为孤儿进程，则会自动被init进程领养父进程接收到子进程退出信号(SIGCHLD)后，利用wait或waitpid主动去回收清理子进程的退出资源**。

实例：

#include <unistd.h>

#include <string.h>

#include <errno.h>

#include <stdlib.h>

#include <stdio.h>

int main(void)

{

pid\_t pid;

pid = fork();

if(pid < 0)

{

fprintf(stderr,"fork error %s\n",strerror(errno));

exit(1);

}else if(pid > 0)

{

while(1)

{

sleep(3);

}

printf("parent pid: %d,ppid: %d\n",getpid(),getppid());

}

else

{

printf("child pid: %d,ppid: %d\n",getpid(),getppid());

exit(1);

}

return 0;

}

#include <unistd.h>

#include <string.h>

#include <errno.h>

#include <stdlib.h>

#include <stdio.h>

int main(void)

{

pid\_t pid;

pid = fork();

if(pid < 0)

{

fprintf(stderr,"fork error %s\n",strerror(errno));

exit(1);

}else if(pid > 0)

{

wait(0);

while(1)

{

sleep(3);

}

printf("parent pid: %d,ppid: %d\n",getpid(),getppid());

}

else

{

printf("child pid: %d,ppid: %d\n",getpid(),getppid());

exit(1);

}

return 0;

}

#### 孤儿进程

父进程退出，而它的一个或多个子进程还在运行，那么子进程将成为孤儿进程。孤儿进程将被 init 进程(1号进程)所收养，并由 init 进程对它们完成退出状态收集清理工作。

实例：

#include <unistd.h>

#include <string.h>

#include <errno.h>

#include <stdlib.h>

#include <stdio.h>

int main(void)

{

pid\_t pid;

pid = fork();

if(pid < 0)

{

fprintf(stderr,"fork error %s\n",strerror(errno));

exit(1);

}else if(pid > 0)

{

sleep(3);

printf("parent pid: %d,ppid: %d\n",getpid(),getppid());

exit(1);

}

else

{

printf("child pid: %d,ppid: %d\n",getpid(),getppid());

}

sleep(5);

printf("child pid: %d,ppid: %d\n",getpid(),getppid());

return 0;

}

### exec函数

在用fork函数创建子进程后，子进程往往要调用一种exec函数以执行另一个程序。

当进程调用一种exec函数时，该进程完全由新程序代换，而新程序则从其main函数开始执行。因为调用exec并不创建新进程，所以前后的进程ID并未改变。exec只是用一个新程序替换了当前进程的征文、数据、堆、和栈段。

#### exec函数家族

#include<unistd.h>

int execl(const char \*path, const char \*arg0, ... /\*, (char \*)0 \*/);

int execv(const char \*path, char \*const argv[]);

int execle(const char \*path, const char \*arg0, ... /\*, (char \*)0, char\*const envp[]\*/);

int execve(const char \*path, char \*const argv[], char \*const envp[]);

int execlp(const char \*file, const char \*arg0, ... /\*, (char \*)0 \*/);

int execvp(const char \*file, char \*const argv[]);

#### system函数

创建一个shell解释器子进程来解释执行其参数

等待进程执行结束

具有信号处理和出错处理

#### 权限管理规则

若进程具有超级用户特权，则setuid函数将实际用户ID，有效用户ID，以及保存的设置用户ID设置为uid。

若进程没有超级用户特权，但是uid等于实际用户ID或保存的设置用户ID，则setuid只将用户ID设置为uid，不改变实际用户ID和保存的设置用户ID。

如果上述条件不满足，则将error设置为EPERM，并返回-1。

### 进程组和守护进程

#### 进程组

进程组由一个或多个共享同一个进程组标识符（PGID）的进程组成。进程组ID是一个数字，其类型与进程ID一样（pid\_t）。一个进程组拥有一个进程组首进程，该进程是创建该组的进程，其进程ID为该进程组的ID，新进程会继承父进程所属的进程组ID。

进程组拥有一个生命周期，其开始时间为首进程创建组的时刻，结束时间为最后一个成员进程退出组的时刻。一个进程可能会因为终止而退出进程组，也可能会因为加入了另一个进程组而退出进程组。进程组首进程无需是最后一个离开进程组的成员。

一个或多个进程的集合通常和作业关联，可以接受同一终端的各种信号

每个进程组有唯一的进程组ID

#include<unistd.h>

pid\_t getpgrp(void)

返回值：调用进程的进程组ID

pid\_t getpgid(pid\_t pid)

返回值：进程pid所进程组的ID，出错返回-1

实例：

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <errno.h>

int main(void)

{

pid\_t pid;

int i = 0;

pid\_t group1;

pid\_t group2;

setpgid(getpid(),getpid());

group1 = getpgid(getpid());

for(i;i < 3;i++)

{

pid = fork();

if(pid < 0)

{

fprintf(stderr,"fork error:%s\n",strerror(errno));

exit(1);

}

else if(pid > 0) //parent process

{

if (i == 0)

{

setpgid(pid,group1);

}

if (i == 1)

{

setpgid(pid,pid);

group2 = getpgid(pid);

}

if (i == 2)

{

setpgid(pid,group2);

}

}else //child process

{

if(i == 0)

{

setpgid(getpid(),group1);

}

if(i == 1)

{

setpgid(getpid(),getpid());

group2 = getpgid(getpid());

}

if(i == 2)

{

setpgid(getpid(),group2);

}

break;

}

}

printf("pid: %d,ppid: %d,groupid: %d\n",getpid(),getppid(),getpgid(0));

sleep(2);

return 0;

}

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <errno.h>

int main(void)

{

pid\_t pid;

int i = 0;

pid\_t group1;

pid\_t group2;

setpgid(getpid(),getpid());

group1 = getpgid(getpid());

for(i;i < 3;i++)

{

pid = fork();

if(pid < 0)

{

fprintf(stderr,"fork error:%s\n",strerror(errno));

exit(1);

}

else if(pid > 0) //parent process

{

if (i == 0)

{

setpgid(pid,group1);

}

if (i == 1)

{

setpgid(pid,group1);

group2 = getpgid(pid);

}

if (i == 2)

{

setpgid(pid,group1);

}

}else //child process

{

if(i == 0)

{

setpgid(getpid(),group1);

}

if(i == 1)

{

setpgid(getpid(),group1);

group2 = getpgid(getpid());

}

if(i == 2)

{

setpgid(getpid(),group1);

}

break;

}

}

printf("pid: %d,ppid: %d,groupid: %d\n",getpid(),getppid(),getpgid(0));

pause();

return 0;

}

#### 会话

会话是一组进程组的集合。进程的会话成员关系是由其会话标识符（SID）确定的，会话标识符与进程组ID一样，是一个类型为pid\_t的数字。会话首进程是创建该新会话的进程，其进程ID会成为会话ID。新进程会继承其父进程的会话ID。

一个会话中的所有进程共享单个控制终端。控制终端会在会话首进程首次打开一个终端设备时被建立。一个终端最多可能会成为一个会话的控制终端。

在任一时刻，会话中的其中一个进程组会成为终端的前台进程组，其他进程组会成为后台进程组。只有前台进程组中的进程才能从控制终端中读取输入。当用户在控制终端中输入其中一个信号生成终端字符之后，该信号会被发送到前台进程组中的所有成员。这些字符包括生成SIGINT的中断字符（通常是Control-C）、生成SIGQUIT的退出字符（通常是Control-\）、生成SIGSTP的挂起字符（通常是Control-Z）。

当到控制终端的连接建立起来（即打开）之后，会话首进程会成为该终端的控制进程。成为控制进程的主要标志是当断开与终端之间的连接时内核会向该进程发送一个SIGHUP信号。

注：通过检查Linux特有的/proc/PID/stst文件，就能确定任意进程的进程组ID和会话ID。

#### 作业

会话和进程组的主要用于是用于shell作业控制。

#### 组长进程

每个进程组可以有个组长进程，组长进程的ID就是进程组的ID

组长进程可以创建进程组以及该组中的进程

组长进程的终止与进程组的是否终止无关

#include<unistd.h>

int setpgid(pid\_t pid,pid\_t pgid);

返回值：成功返回0，出错返回-1。

#### 前台进程组

前台进程组和控制终端相关联，由控制终端产生的信号均发送给前台进程组

#include<unistd.h>

pid\_t tcgetpgrp(int filedes);

返回值：若成功返回前台进程组的ID，出错返回-1

int tcsetpgrp(int filedes,pid\_t pgrpid);

返回值：成功返回0，出错返回-1.

#### 守护进程

守护进程是生存期长的一种进程。他们常常在系统引导装入时启动，在系统关闭时终止

所有守护进程都以超级用户（用户ID为0）的优先权运行

守护进程没有控制终端

守护进程的父进程都是init进程

守护进程编程过程

使用umask将文件模式创建屏蔽字设为0

调用fork，然后让父进程退出(exit)

调用setsid创建一个新会话

将当前工作目录更改为根目录

关闭不需要的文件描述符

## 进程调度

调度程序负责决定将哪个进程投入运行，何时运行以及运行多长时间。进程调度程序（常常简称调度程序）可看做在可运行态进程之间分配有限的处理器时间资源的内核子系统。调度程序是像Linux这样的多任务操作系统的基础。只有通过调度程序的合理调度，系统资源才能最大限度发挥作用，多进程才会有并发执行的效果。

调度程序没有太复杂的原理。最大限度地利用处理器时间的原则是，只要有可以执行的进程，那么就总会有进程在执行。但是只要系统中可运行的进程的数目比处理器的个数多，就注定某一给定时刻会有一些进程不能执行。这些进程在等待运行。在一组处于可运行状态的进程中选择一个来执行，是调度程序所需要完成的基本工作。

任何进程要想占有CPU，从而真正处于执行状态，就必须经由进程调度。进程调度机制主要涉及到调度方式、调度时机和调度策略。

### 多任务

多任务操作系统就是能够同时并发地交互执行多个进程的操作系统。

多任务系统可以划分为两类：非抢占式多任务（cooperative multitasking）和抢占式多任务（preemptive multitasking）。

Linux提供了抢占式的多任务模式，在此模式下，由调度程序来决定什么时候停止一个进程的运行，以便其他进程能够得到执行机会。这个强制的挂起动作就叫做抢占（preemption）。进程在被抢占之前能够运行的时间是预先设置好的，叫做进程的时间片（timeslice）。时间片实际上就是分配给每个可运行进程的处理器时间段。有效管理时间片能使调度程序从系统全局的角度做出调度决定，这样做还可以避免个别进程独占系统资源。当今众多现代操作系统对程序运行都采用了动态时间片计算的方式，并且引入了可配置的计算策略。不过我们将看到，Linux独一无二的“公平”调度程序本身并没有采用时间片来达到公平调度。

在非抢占式多任务模式下，除非进程自己主动停止运行，否则它会一直执行。进程主动挂起自己的操作称为让步（yielding）。

### 调度方式

Linux内核的调度方式基本上采用“抢占式优先级”方式，即当进程在用户模式下运行时，不管是否自愿，在一定条件下(如时间片用完或等待I/O)，核心就可以暂时剥夺其运行而调度其它进程进入运行。但是，一旦进程切换到内核模式下运行，就不受以上限制而一直运行下去，直至又回到用户模式之前才会发生进程调度。

Linux系统中的调度策略基本上继承了Unix的以优先级为基础的调度。就是说，核心为系统中每个进程计算出一个优先权，该优先权反映了一个进程获得CPU使用权的资格，即高优先权的进程优先得到运行。核心从进程就绪队列中挑选一个优先权最高的进程，为其分配一个CPU时间片，令其投入运行。在运行过程中，当前进程的优先权随时间递减，这样就实现了“负反馈”作用：经过一段时间之后，原来级别较低的进程就相对“提升”了级别，从而有机会得到运行。当所有进程的优先权都变为0时，就重新计算一次所有进程的优先权。

### 调度策略

#### I/O消耗型和处理器消耗型进程

进程可以被分为I/O消耗型和处理器消耗型。前者指进程的大部分时间用来提交I/O请求或是等待I/O请求。因此，这样的进程经常处于可运行状态，但是通常都是短短的一会儿，因为它在等待更多的I/O请求时最后总会阻塞（这里所说的I/O是指任何类型的可阻塞资源，比如键盘输入，或者是网络I/O）。举例来说，多数用户图形界面程序（GUI）都是属于I/O密集型，即便它们从来不读取或者写入磁盘，它们也会在多数时间里都在等待来自鼠标或者键盘的用户交互操作。

相反，处理器耗费型进程把时间大多用在执行代码上。除非被抢占，否则它们通常都一直不停地运行，因为它们没有太多的I/O需求。但是，因为它们不属于I/O驱动类型，所以从系统响应速度考虑，调度器不应该经常让它们运行。对于这类处理器消耗型的进程，调度策略往往是尽量降低它们的调度频率，而延长其运行时间。处理器消耗型进程的极端例子就是无限循环地执行。更具代表性的例子是那些执行大量数学计算的程序，如MATLAB。

调度策略通常要在两个矛盾的目标中间寻找平衡：进程响应迅速（响应时间短）和最大系统利用率（高吞吐量）。为了满足上述需求，调度程序通常采用一套非常复杂的算法来决定最值得运行的进程投入运行，但是它往往并不保证低优先级进程会被公平对待。UNIX系统的调度程序更倾向于I/O消耗型程序，以提供更好的程序响应速度。Linux为了保证交互式应用和桌面系统的性能，所以对进程的响应做了优化（缩短响应时间），更倾向于优先调度I/O消耗型进程。虽然如此，但在下面不会看到，调度程度也并未忽略处理器消耗型的进程。

#### 进程优先级

调度算法中最基本的一类就是基于优先级的调度。通常做法是优先级高的进程先运行，低的后运行，相同优先级的进程按轮转方式进行调度。

Linux采用两个不同的优先级范围。第一种是nice值，范围是-20~19，默认为0，越大表示优先级越低。第二种范围是实时优先级，其值是可以配置的，变化范围为0~99，值越高优先级越高。

#### 时间片

时间片是一个数值，表示进程在被抢占前所能持续运行的时间。

时间片过长会导致系统对交互的响应表现欠佳，让人觉得系统无法并发执行应用程序；时间片太短会明显增大进程切换带来的处理器耗时，因为肯定会有相当一部分系统时间用在进程切换上，而这些进程能够用来运行的时间片却更短。此外，I/O消耗型和处理器消耗型的进程之间的矛盾在这里也再次暴露出来：I/O消耗型不需要长的时间片，而处理器消耗型的进程则希望越唱越好（比如这样可以让它们的高速缓存命中率更高）。

从上面的争论中可以看出，任何长时间片都将导致系统交互表现欠佳。很多操作系统中都特别重视这一点，所以默认的时间片很短，如10ms。但是，Linux的CFS调度器并没有直接分配时间片到进程，它是将处理器的使用比例分给了进程。这样一来，进程所获取的处理器时间其实是和系统负载密切相关的。这个比例进一步还会受进程nice值的影响，nice值作为权重将调整进程所使用的处理器时间使用比。具有更高nice值（更低优先权）的进程将被赋予高权重，从而抢得更多的处理器使用比。

当一个进程进入可运行态，它就被准许投入运行。在多数操作系统中，是否要将一个进程立刻投入运行（也就是抢占当前进程），是完全由进程优先级和是否有时间片决定的。而**在Linux中使用新的CFS调度器，其抢占时机取决于新的可运行程序消耗了多少处理器使用比**。如果消耗的使用比比当前进程小，则新进程立刻投入运行抢占当前进程。否则，将推迟其运行。

#### 实时调度策略

Linux系统针对不同类别的进程提供了三种不同的调度策略，即**SCHED\_FIFO、SCHED\_RR及SCHED\_OTHER**。

SCHED\_FIFO适合于实时进程，它们对时间性要求比较强，而每次运行所需的时间比较短，一旦这种进程被调度开始运行后，就要一直运行到自愿让出CPU，或者被优先权更高的进程抢占其执行权为止。

SCHED\_RR对应“时间片轮转法”，适合于每次运行需要较长时间的实时进程。一个运行进程分配一个时间片(如200毫秒)，当时间片用完后，CPU被另外进程抢占，而该进程被送回相同优先级队列的末尾。**SCHED\_OTHER是传统的Unix调度策略，适合于交互式的分时进程。**这类进程的优先权取决于两个因素，一个因素是进程剩余时间配额，如果进程用完了配给的时间，则相应优先权为0；另一个是进程的优先数nice，这是从Unix系统沿袭下来的方法，优先数越小，其优先级越高。

**nice的取值范围是-19－20**。用户可以利用nice命令设定进程的nice值。但一般用户只能设定正值，从而主动降低其优先级；只有特权用户才能把nice的值置为负数。进程的优先权就是以上二者之和。核心动态调整用户态进程的优先级。这样，一个进程从创建到完成任务后终止，需要经历多次反馈循环。当进程再次被调度运行时，它就从上次断点处开始继续执行。对于实时进程，其优先权的值是(1000＋设定的正值)，因此，至少是1000。所以，实时进程的优先权高于其它类型进程的优先权。另外，时间配额及nice值与实时进程的优先权无关。如果系统中有实时进程处于就绪状态，则非实时进程就不能被调度运行，直至所有实时进程都完成了，非实时进程才有机会占用CPU。

后台命令（在命令末尾有&符号，如gcc f1.c& )对应后台进程（又称后台作业），后台进程的优先级低于任何交互（前台）进程的优先级。所以，只有当系统中当前不存在可运行的交互进程时，才调度后台进程运行。后台进程往往按批处理方式调度运行。

### 调度时机

核心进行进程调度的时机有以下几种情况：

1. 当前进程调用系统调用nanosleep( )或pause( )使自己进入睡眠状态，主动让出一段时间的CPU使用权；
2. 进程终止，永久地放弃对CPU的使用；
3. 在时钟中断处理程序执行过程中，发现当前进程连续运行的时间过长；
4. 当唤醒一个睡眠进程时，发现被唤醒的进程比当前进程更有资格运行；
5. 一个进程通过执行系统调用来改变调度策略或降低自身的优先权(如nice命令)，从而引起立即调度。

### 调度算法

进程调度的算法应该比较简单，以便减少频繁调度时的系统开销。Linux执行进程调度时，首先查找所有在就绪队列中的进程，从中选出优先级最高且在内存的一个进程。如果队列中有实时进程，那么实时进程将优先运行。如果最需要运行的进程不是当前进程，那么当前进程就被挂起，并且保存它的现场所涉及的一切机器状态，包括计数器和CPU寄存器等，然后为选中的进程恢复运行现场。

参考：

<https://blog.csdn.net/qq_43414142/article/details/90676984>

<https://www.cnblogs.com/alantu2018/p/8460451.html>

#### 先来先服务（FCFS)调度算法

#### 短作业优先（SJF）调度算法

#### 优先调度算法

#### 高响应比优先调度算法

#### 时间片轮转调度算法

#### 多级反馈队列调度算法

### CFS算法实现

#### 时间记账

#### 进程选择

CFS利用一个简单的规则去均衡进程的虚拟运行时间：当CFS需要选择下一个运行进程时，它会挑一个具有最小vruntime的进程。这就是CFS带哦度算法的核心：选择具有最小vruntime的任务。

CFS使用红黑树来组织可运行进程队列，并利用其迅速找到最小vruntime值的进程。

#### 调度器入口

#### 睡眠和唤醒

### 抢占和上下文切换

进程上下文是进程执行活动全过程的静态描述。我们把已执行过的进程指令和数据在相关寄存器与堆栈中的内容称为进程上文，把正在执行的指令和数据在寄存器与堆栈中的内容称为进程正文，把待执行的指令和数据在寄存器与堆栈中的内容称为进程下文。

实际上linux内核中，进程上下文包括进程的虚拟地址空间和硬件上下文。

进程硬件上下文包含了当前cpu的一组寄存器的集合，arm64中使用task\_struct结构的thread成员的cpu\_context成员来描述，包括x19-x28,sp, pc等。

注：可以利用绑核来减少CPU之间切换。

#### 用户抢占

#### 内核抢占

### 与调度相关系统调用

#### 与调度策略和优先级相关的系统调用

#### 与处理器绑定有关的系统调用

#### 放弃处理器时间

## 僵尸进程

进程停止后，该进程就会从进程表中移除。但是，有时候有些程序即使执行完了也依然留在进程表中。那么，这些完成了生命周期但却依然留在进程表中的进程，我们称之为“僵尸进程”。

### 产生

当你运行一个程序时，它会产生一个父进程以及很多子进程。所有这些子进程都会消耗内核分配给它们的内存和CPU资源。

这些子进程完成执行后会发送一个exit信号然后死掉。这个exit信号需要被父进程所读取。父进程需要随后调用 wait 命令来读取子进程的退出状态，并将子进程从进程表中移除。

若父进程正确第读取了子进程的exit信号，则子进程会从进程表中删掉。

但若父进程未能读取到子进程的exit信号，则这个子进程虽然完成执行处于死亡的状态，但也不会从进程表中删掉。

### 危害

由于僵尸进程并不做任何事情，不会使用任何资源也不会影响其它进程，因此存在僵尸进程也没什么坏处。不过由于进程表中的退出状态以及其它一些进程信息也是存储在内存中的，因此存在太多僵尸进程有时也会是一些问题。

你可以想象成这样：

“你是一家建筑公司的老板。你每天根据工人们的工作量来支付工资。 有一个工人每天来到施工现场，就坐在那里， 你不用付钱， 他也不做任何工作。 他只是每天都来然后呆坐在那，仅此而已！”

这个工人就是僵尸进程的一个活生生的例子。但是， 如果你有很多僵尸工人， 你的建设工地就会很拥堵从而让那些正常的工人难以工作。

### 查找

打开终端并输入下面命令:

**ps aux | grep Z**

会列出进程表中所有僵尸进程的详细内容。

### 杀死

如何杀掉僵尸进程？

正常情况下我们可以用SIGKILL信号来杀死进程，但是僵尸进程已经死了，你不能杀死已经死掉的东西。 因此你需要输入的命令应该是

**kill -s SIGCHLD pid**

将这里的pid替换成父进程的进程id，这样父进程就会删除所有以及完成并死掉的子进程了。

你可以把它想象成：

“你在道路中间发现一具尸体，于是你联系了死者的家属，随后他们就会将尸体带离道路了。”

不过许多程序写的不是那么好，无法删掉这些子僵尸（否则你一开始也见不到这些僵尸了）。 因此确保删除子僵尸的唯一方法就是杀掉它们的父进程。

# 线程

## 背景



进程的目的之一是为了提供并发能力，但进程的创建需要新分配虚拟地址空间、页表、物理内存等等，那么我们是否可以创建轻量级的并发实体，共享进程的资源？这就引入线程。

并发实体尽可能去共享进程的资源，比如共享一块地址空间，共享一个页表和一块物理内存。

**线程产生的原因：**

进程可以使多个程序能并发执行，以提高资源的利用率和系统的吞吐量；但是其具有一些缺点：

1、进程在同一时间只能干一件事；

2、进程在执行的过程中如果阻塞，整个进程就会挂起，即使进程中有些工作不依赖于等待的资源，仍然不会执行。

因此，操作系统引入了比进程粒度更小的线程，作为并发执行的基本单位，从而减少程序在并发执行时所付出的时空开销，提高并发性。和进程相比，线程的优势如下：

从资源上来讲，线程是一种非常"节俭"的多任务操作方式。在linux系统下，启动一个新的进程必须分配给它独立的地址空间，建立众多的数据表来维护它的代码段、堆栈段和数据段，这是一种"昂贵"的多任务工作方式。

从切换效率上来讲，运行于一个进程中的多个线程，它们之间使用相同的地址空间，而且线程间彼此切换所需时间也远远小于进程间切换所需要的时间。据统计，一个进程的开销大约是一个线程开销的30倍左右。

从通信机制上来讲，线程间方便的通信机制。对不同进程来说，它们具有独立的数据空间，要进行数据的传递只能通过进程间通信的方式进行，这种方式不仅费时，而且很不方便。线程则不然，由于同一进城下的线程之间共享数据空间，所以一个线程的数据可以直接为其他线程所用，这不仅快捷，而且方便。

除以上优点外，多线程程序作为一种多任务、并发的工作方式，还有如下优点：

1、使多CPU系统更加有效。操作系统会保证当线程数不大于CPU数目时，不同的线程运行于不同的CPU上。

2、改善程序结构。一个既长又复杂的进程可以考虑分为多个线程，成为几个独立或半独立的运行部分，这样的程序才会利于理解和修改。

## 概述

进程时系统中**程序执行和资源分配**的基本单元，每个进程都有自己的数据段、代码段、堆栈段。

线程，有时也被称为轻量级进程，是**程序执行流的最小单元**，是进程中的一个实体，是被系统独立调度和分派的基本单位。与进程不同，**线程与资源分配无关，线程自己不拥有系统资源，它属于某一个进程，并与进程内的其他线程一起共享进程的资源。线程只由相关堆栈（系统栈或用户栈）寄存器和线程控制表TCB组成。**

**总结：**

进程时资源管理的最小单元；线程是程序执行的最小单元。

进程作为资源分配的基本单元；线程作为调度和分配的基本单元。

### 内存布局

共享进程的内存布局。

线程共享的环境包括：进程代码段、进程的公有数据（利用这些共享的数据，线程很容易的实现相互之间的通讯）、进程打开的文件描述符、信号的处理器、进程的当前目录和进程用户ID与进程组ID。

每个线程私有的资源包括：

1. 线程ID
2. 寄存器组的值
3. 线程的栈
4. 错误返回码errno值
5. 线程的信号屏蔽码
6. 线程的优先级

### 线程标识

每个进程内部的不同线程都有自己的唯一标识

线程标识（ID）只在它所属的进程环境中有效

线程标识是pthread\_t数据类型

#include<pthread.h>

int pthread\_equal(pthread\_t,pthread\_t)

返回值：相等返回非0，否则返回0

pthread\_t pthread\_self(void)

返回值：调用线程的线程ID

### 状态转换



## 分类

在许多类Unix系统中，如Linux、FreeBSD、Solaris等，进程一直都是操作系统内核调用的最小单位，也都采用多进程模型。后来引入了线程概念，有以下两种概念的线程：

### 用户级线程

用户级线程(User-Level Thread，ULT)。由应用进程利用线程库创建和管理，不在内核中实现线程，只在用户态中模拟出多线程，不依赖于核心，操作系统内核完全不知道多线程的存在。

主要解决的是上下文切换问题，其调度算法和调度过程全部由用户决定。

### 内核线线程

内核线线程(Kernel-Level Thread，KLT)，又称为内核支持的线程或轻量级进程。是在核心空间实现的，内核为每个线程在核心空间中设置了一个线程控制块，用来登记该线程的线程标识符、值、状态、优先级等信息，所有对线程的操作，如创建、撤销和切换都是通过系统功能调用由内核中的相应处理完成，内核维护进程及线程的上下文切换以及线程切换，类系统中一般通过修改进程的实现方式来实现，可以使用不完全的进程创建方式创建共享数据空间的进程，在Linux下这种系统调用为clone()，而在FreeBSD下它为rfork()。

现在大多数操作系统都采用用户级线程和内核级线程并存的方法。用户级线程可与内核线程实现“一对一”，“多对一”的对应关系。

## Linux线程实现

线程的实现方式：

### LWP

1. LWP（轻量级进程）作为多线程方案

纯用户空间多线程方案



**优点：**

真正实现并发操作；

克服阻塞问题。

**缺点：**

控制转移开销大，调度算法由操作系统核心确定，应用进程无法影响线程的切换；

数量存在限制。

### 协程

1. 纯用户空间多线程方案（协程方案）



**优点：**

切换速度快；

调度算法可专用（用户态调度）；

可运行在任何操作系统上（汇编处理）。

**缺点：**

阻塞问题（多个用户态U对应一个内核态K）；

多处理器利用问题；

如果某个用户被阻塞，导致进程同样被阻塞。

调度实体（Linux为例）：

进程：使用PCB描述-->进程控制块；操作系统感知其存在，来源于PCB（结构体），每个进程都需要实例化一个PCB。

线程：实际上也是用PCB描述，只是PCB的某些变量处理和进程有区别。

### 混合

1. 混合版多线程方案



用户线程底层对应LWP，减少线程切换代价的同时，提升并行能力（GO语言协程多核并发方案）。

以下线程均为用户级线程。在Linux中，一般采用Pthread线程库实现线程的访问与控制，由POSIX提出，具有良好的可移植性。

Linux线程程序编译需要在gcc上链接库pthread。

## 状态



## 线程操作

### 线程创建

#include<pthread.h>

int pthread\_create(pthread\_t \*restrict tidp,const ptread\_attr\_t

\*restrict attr,void \*(start\_rtn)(void),void\*restrict arg);

返回：成功返回0，否则返回错误编号。

参数：

tidp：线程标识符

attr：线程属性设置

start\_rtn：线程函数起始地址

arg：传递给start\_rtn函数的地址开始运行

在编译时注意加上‐lpthread参数，以调用链接库。因为pthread并非Linux系统的默认库，而是posix线程库，在Linux中将其作为一个库来使用，因此加上 -lpthread（或‐pthread）以显示的链接该库。函数在执行错误时的错误信息将作为返回值返回。

### 启动线程

### 执行线程

### 线程中断

### 线程休眠

### 线程等待/通知

### 线程挂起/恢复

### 线程植入/让步

### 线程终止和清理处理

线程退出方式：

1、线程从启动例程返回，返回值是线程的退出码

2、线程可以被同一进程的其他线程取消

3、线程自己调用pthread\_exit

#### pthread\_exit

#include<pthread.h>

void pthread\_exit(void \*retval);

int phread\_join(pthread\_t \*th,void \*\*thread\_return);

返回值：成功返回0，否则返回错误编码

注：pthread\_exit是线程主动退出，而pthread\_join是以阻塞的方式等待线程退出。

pthread\_exit：

retval：phread\_exit调用者线程的返回值，可由其他函数和pthread\_join来检测获取线程退出时使用函数pthread\_exit，是线程的主动行为。

由于一个进程中的多个线程共享数据段，因此通常在线程退出后，退出线程所占用的资源并不会随线程结束而释放。所以需要pthread\_join函数来等待线程结束，类似于wait系统调用。

#### pthread\_jion

int phread\_join(pthread\_t \*th,void \*\*thread\_return);

返回值：成功返回0，否则返回错误编码

pthread\_jion：

th：等待线程的标识符

thread\_return：用户定义指针，用来存储被等待线程的返回值。

实例：

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <string.h>

typedef struct pth\_arg

{

int start;

int end;

}pth\_arg;

void \*th\_fun(void \*arg)

{

pth\_arg \*pa = (pth\_arg\*)arg;

int s = pa->start;

int e = pa->end;

int i;

int sum = 0;

for(i = s;i <= e;i++)

{

sum += i;

}

//return (void\*)sum;

pthread\_exit((void\*)sum);

}

int main(void)

{

pthread\_t pth1;

int err;

pth\_arg arg1 = {1,100};

if((err = pthread\_create(&pth1,NULL,th\_fun,(void\*)&arg1)) != 0)

{

fprintf(stderr,"pthread\_create:%s\n",strerror(err));

exit(1);

}

int \*ret;

pthread\_join(pth1,(void\*\*)&ret);

printf("sum: %d\n",(int)ret);

printf("finish\n");

return 0;

}

#### pthread\_cancel

#include <pthread.h>

int pthread\_cancel(pthread\_t pid)

线程可以通过pthread\_cancel取消其他线程

被终止线程可以自己选择忽略取消或控制取消

#### pthread\_cleanup\_push

#### pthread\_cleanup\_pop

#include <pthread.h>

void pthread\_cleanup\_push(void\*(rtn)(void\*),void\* arg);

void pthread\_cleanup\_pop(int execute);

触发线程调用清理处理函数动作：

1、调用pthread\_exit

2、响应取消请求

3、用非零execute参数调用thread\_cleanup\_pop时

实例：

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <string.h>

void clean\_fun(void \*arg)

{

printf("clead arg:%s\n",(char\*)arg);

}

void \*th\_fun(void \*arg)

{

printf("id: %ld\n",pthread\_self());

int execute = (int)arg;

pthread\_cleanup\_push(clean\_fun,"hello world");

pthread\_cleanup\_push(clean\_fun,"hello jikexueyuan");

pthread\_cleanup\_pop(execute);

pthread\_cleanup\_pop(execute);

pthread\_exit((void\*)1);

}

int main(void)

{

pthread\_t pth1;

pthread\_t pth2;

int err;

if((err = pthread\_create(&pth1,NULL,th\_fun,(void\*)0)) != 0)

{

fprintf(stderr,"pthread\_create: %s\n",strerror(err));

exit(1);

}

pthread\_join(pth1,NULL);

printf("----------------------\n");

if((err = pthread\_create(&pth2,NULL,th\_fun,(void\*)1)) != 0)

{

fprintf(stderr,"pthread\_create: %s\n",strerror(err));

exit(1);

}

pthread\_join(pth2,NULL);

return 0;

}

## 线程分组

## 守护线程

## 线程优先级

## 线程属性操作

### 线程属性操作

#### 数据类型

pthread\_attr\_t

#### 属性 (posix.1)

分离状态属性

栈末尾警戒缓冲区大小

线程栈的最低地址

线程栈的大小

线程绑定属性

#### 线程属性

线程属性操作：

pthread\_attr\_init 初始化属性的函数

pthread\_attr\_setscope 设置绑定属性

pthread\_attr\_setdetachstate 设置分离属性

pthread\_attr\_getschdparm 获取线程优先级

pthread\_attr\_setschedparam 设置线程优先级

#### 属性初始化和销毁

#include <phtread.h>

int pthread\_attr\_init(pthread\_attr\_t \*attr)

int pthread\_attr\_destroy(pthread\_attr\_t \*attr)

返回：成功返回0，否则返回错误编号。

#### 分离属性

线程属性-分离属性

#incldue <pthread.h>

int pthread\_attr\_getdetachstat(const pthread\_attr\_t \*attr,int \*detachstate);

int pthread\_attr\_setdetachstat(const pthread\_attr\_t \*attr,int \*detachstate);

返回值：成功返回0，出错返回错误编码。

detachstate取值

PTHREAD\_CREATE\_JOINABLE 正常启动线程

PTHREAD\_CREATE\_DETACHED 以分离状态启动线程

实例：

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <string.h>

void out\_attr(pthread\_attr\_t attr)

{

int stat;

int err;

if((err=pthread\_attr\_getdetachstate(&attr,&stat)) != 0)

{

fprintf(stderr,"%s\n",strerror(err));

exit(1);

}else

{

if(stat == PTHREAD\_CREATE\_JOINABLE)

{

printf("attr: PHTREAD\_CREATE\_JOINABLE\n");

}else if(stat == PTHREAD\_CREATE\_DETACHED)

{

printf("attr: PTHREAD\_CREATE\_DETACHED\n");

}else

{

printf("attr error!\n");

}

}

}

void \*th\_fun(void \*arg)

{

int i = 1;

int sum = 0;

for(i;i <= 100;i++)

{

sum += i;

}

return (void\*)sum;

}

int main(void)

{

pthread\_t pth1;

pthread\_t pth2;

pthread\_attr\_t attr;

pthread\_attr\_init(&attr);

out\_attr(attr);

int err;

int ret;

if((err=pthread\_create(&pth1,&attr,th\_fun,(void\*)0)) != 0)

{

fprintf(stderr,"%s\n",strerror(err));

exit(1);

}

if(pthread\_join(pth1,(void\*)&ret) != 0)

{

fprintf(stderr,"%s\n",strerror(err));

exit(1);

}else

{

printf("ret sum：%d\n",ret);

}

printf("...................detach pthread.............\n");

if((err=pthread\_attr\_setdetachstate(&attr,PTHREAD\_CREATE\_DETACHED)) != 0)

{

fprintf(stderr,"%s\n",strerror(err));

exit(1);

}

if((err=pthread\_create(&pth2,&attr,th\_fun,(void\*)0)) != 0)

{

fprintf(stderr,"%s\n",strerror(err));

exit(1);

}

if((err=pthread\_join(pth2,(void\*)&ret)) != 0)

{

fprintf(stderr,"%s\n",strerror(err));

exit(1);

}else

{

printf("ret sum:%d\n",ret);

}

if((err=pthread\_attr\_destroy(&attr)) != 0)

{

fprintf(stderr,"%s\n",strerror(err));

exit(1);

}

return 0;

}

#### 绑定

绑定是指某个线程固定的“绑”在一个轻进程上。被绑定的线程具有较高的相应速度，这是因为CPU时间片的调度室面向轻进程的，绑定的线程可以保证在需要的时候他总有一个轻进程可用。

通过设置被绑定的轻进程的优先级和调度级可以使得绑定的线程满足诸如实时反应之类的要求。

#include<pthread.h>

pthread\_attr\_setscope(pthread\_attr\_t \*attr,init scope)；

返回值：成功0，错误-1。

scope取值：

PTHREAD\_SCOPE\_SYSTEM（绑定）

PTHREAD\_SCOPE\_PRCESS（非绑定）

### 互斥锁

#### 概念

mutex是一种简单的加锁的方法来控制对共享资源的访问。在同一时刻只能有一个线程掌握某个互斥上的锁，拥有上锁状态的线程能够对共享资源进行访问。若其他线程希望上锁一个已经被上了互斥锁的资源，则该线程挂起，知道上锁的线程释放互斥锁为止。

#### 数据类型

互斥锁数据类型:

pthread\_mutex\_t

#### 分类

互斥锁可分为以下几种：

标准互斥锁：PTHREAD\_MUTEX\_NORMAL

递归互斥锁：PTHREAD\_MUTEX\_RECURSIVE

检错互斥锁：PTHREAD\_MUTEX\_ERRORCHECK

默认互斥锁：PTHREAD\_MUTEX\_DEFAULT

#### 互斥锁属性

##### 进程共享属性

进程共享属性决定了不同进程中的线程在进程的共享区中分配的互斥量是否可以用于进程同步

PTHREAD\_PROCESS\_PRIVATE(默认情况)

PTHREAD\_PROCESS\_SHARED(进程间同步)

进程共享操作

#include <pthread.h>

int pthread\_mutexattr\_getpshared(const pthread\_mutexattr\_t \*attr,int

\*pshared);

int pthread\_mutexattr\_setpshared(pthread\_mutexattr\_t \*attr,int pshared);

返回值：成功返回0，出错返回错误编号。

##### 类型属性

#### 互斥锁操作

##### 创建/销毁互斥锁

#include <pthread.h>

int pthread\_mutex\_init(pthread\_mutex\_t \*mutex,const pthread\_mutex\_attr

\*mutexattr);

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

参数mutexattr取值：

PTHREAD\_MUTEX\_INITIALIZER 创建快速互斥锁

PTHREAD\_RECURSIVE\_MUTEX\_INITIALIZER\_NP 创建递归互斥锁

PTHREAD\_REEORCHECK\_MUTEX\_INITIALIZER\_NP 创建检错互斥锁

##### 上锁和解锁

#include <pthread.h>

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

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

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

返回值：成功返回0，出错返回出错码。

#### 互斥锁属性操作

##### 创建/销毁属性

#include <pthread.h>

int pthread\_mutexattr\_init(pthread\_mutexattr\_t \*attr);

int pthread\_mutexattr\_destroy(pthread\_mutexattr\_t \*attr);

返回值：成功返回0，出错返回错误编码。

##### 互斥锁类型操作

#include <pthread.h>

int pthread\_mutexattr\_gettype(const pthread\_mutexattr\_t \*attr,int

\*type);

int pthread\_mutexattr\_settype(pthread\_mutexattr\_t \*attr,int type);

返回值：成功返回0，出错返回错误编号。

### 读写锁

线程使用互斥锁缺乏读并发性。

当读操作较多，写操作较少时，可使用读写锁提高线程读并发性

#### 数据类型

读写锁数据类型

pthread\_rwlock\_t

读写锁属性数据类型

pthread\_rwlockattr\_t

#### 读写锁操作

##### 创建/销毁锁

#include <pthread.h>

int pthread\_rwlock\_init(pthread\_rwlock\_t \*rwlock, const

pthread\_rwlockattr\_t \*attr);

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

返回值：成功返回0，出错返回错误编号。

##### 加锁和解锁

#include <pthread.h>

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

Int pthead\_rwlock\_wdlock(pthread\_rwlock\_t \*rwlock);

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

返回值：成功返回0，出错返回错误编号。

#### 读写锁属性操作

##### 创建/销毁属性

#include <pthread.h>

int pthread\_rwlockattr\_init(pthread\_rwlockattr\_t \*attr);

int pthread\_rwlockattr\_destroy(pthread\_rwlockattr\_t \*attr);

返回值：成功返回0，出错返回错误编号。

##### 进程共享属性

#include <pthread.h>

int pthread\_rwlockattr\_getpshared(const pthread\_rwlockattr\_t \*attr,int

pshared);

int pthread\_rwlockattr\_setpshared(pthread\_rwlockattr\_t \*attr,int

pshared);

返回值：成功返回0，出错返回错误编号。

### 条件变量

条件变量解决了线程同步问题

条件变量允许线程等待特定条件发生

#### 数据类型

条件变量数据类型：

pthread\_cond\_t

条件变量属性数据类型

pthread\_condattr\_t

静态条件变量可使用PTHREAD\_COND\_INITIALIZER

#### 条件变量操作

##### 创建和销毁

#include <ptherad.h>

int pthread\_cond\_init(pthread\_cond\_t \*cond,pthread\_condattr\_t \*attr);

int pthread\_cond\_destroy(pthread\_cond\_t \*cond);

返回值：成功返回0，出错返回错误编号。

##### 等待

条件变量操作---等待

#incldue <pthread.h>

int pthread\_cond\_wait(pthread\_cond\_t \*cond,pthread\_mutex\_t \*mutex);

int pthread\_cond\_timewait(pthread\_cond\_t \*cond,pthread\_mutex\_t

\*mutex,const struct timespec \*timeout);

返回值：成功返回0，出错返回错误编号。

struct timesec{

time\_t tv\_sec; /\*seconds\*/

long tv\_nsec; /\*nanoseconds\*/

}

##### 通知

条件变量操作--通知

#include <pthread.h>

int pthread\_cond\_signal(pthread\_cond\_t \*cond);

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

返回值：成功返回0，出错返回错误编号。

#### 条件变量属性

##### 创建和销毁

条件变量属性操作--创建和销毁

#include<pthread.h>

int pthread\_condattr\_init(pthread\_condattr\_t \*attr);

init pthread\_condattr\_destroy(pthread\_condattr\_t \*attr);

返回值：成功返回0，出错返回错误编号。

##### 共享属性

条件变量属性--进程共享属性

#include <pthread.h>

int pthread\_condattr\_getpshared(const pthread\_condattr\_t \*attr,int

\*pshared);

init pthread\_condattr\_setpshared(pthread\_condattr\_t \*attr,int pshared);

返回值：成功返回0，出错返回错误编号。

## 多线程编程



### 背景

**为什么要多线程？**

在我们真实业务中，我们是什么流程？



上图的流程：

1、先发起网络请求

2、Web服务器解析请求

3、请求后端的数据库获取数据

4、获取数据后，进行处理

5、把处理结果放回给用户

这个是我们处理业务的时候，常规的请求流程；我们看一下整个过程涉及到什么计算机处理。

1、网络请求----->网络IO

2、解析请求----->CPU

3、请求数据库----->网络IO

4、MySQL查询数据----->磁盘IO

5、MySQL返回数据----->网络IO

6、数据处理----->CPU

7、返回数据给用户----->网络IO

在真实业务中我们不单单会涉及CPU计算，还有网络IO和磁盘IO处理，这些处理是非常耗时的。如果一个线程整个流程是上图的流程，真正涉及到CPU的只有2个节点，其他的节点都是IO处理，那么线程在做IO处理的时候，CPU就空闲出来了，CPU的利用率就不高。

现在知道多线程的用处：就是为了提升CPU利用率。

### 概述

多线程意味着你能够在同一个应用程序中运行多个线程，我们知道，指令是在CPU中执行的，多线程应用程序就像是具有多个CPU在同时执行应用程序的代码。



其实这是一种假象，线程数量并不等于CPU数量，单个CPU将在多个线程之间共享CPU的时间片，在给定的时间片内执行每个线程之间的切换，每个线程也可以由不同的 CPU 执行，如下图所示：



### 并发

#### 并发VS并行

并发：一个处理器同时处理多个任务

并行：多个处理器或者是多核的处理器同时处理多个不同的任务

并发意味着应用程序会执行多个的任务，但是如果计算机只有一个 CPU 的话，那么应用程序无法同时执行多个的任务，但是应用程序又需要执行多个任务，所以计算机在开始执行下一个任务之前，它并没有完成当前的任务，只是把状态暂存，进行任务切换，CPU 在多个任务之间进行切换，直到任务完成。如下图所示：



并行是指应用程序将其任务分解为较小的子任务，这些子任务可以并行处理，例如在多个CPU上同时进行。



#### 同步VS异步

**同步和异步的区别：**

同步：执行某个操作开始后就一直等着按部就班的直到操作结束

异步：执行某个操作后立即离开，后面有响应的话再来通知执行者

接着我们再了解一个重要的概念：

临界区：公共资源或者共享数据

由于共享数据的出现，必然会导致竞争，所以我们需要再了解一下：

阻塞：某个操作需要的共享资源被占用了，只能等待，称为阻塞

非阻塞：某个操作需要的共享资源被占用了，不等待立即返回，并携带错误信息回去，期待重试

如果两个操作都在等待某个共享资源而且都互不退让就会造成死锁：

死锁：参考著名的哲学家吃饭问题

饥饿：饥饿的哲学家等不齐筷子吃饭

活锁：相互谦让而导致阻塞无法进入下一步操作，跟死锁相反，死锁是相互竞争而导致的阻塞

#### 并发级别

理想情况下我们希望所有线程都一起并行飞起来。但是CPU数量有限，线程源源不断，总得有个先来后到，不同场景需要的并发需求也不一样，比如秒杀系统我们需要很高的并发程度，但是对于一些下载服务，我们需要的是更快的响应，并发反而是其次的。所以我们也定义了并发的级别，来应对不同的需求场景。

阻塞：阻塞是指一个线程进入临界区后，其它线程就必须在临界区外等待，待进去的线程执行完任务离开临界区后，其它线程才能再进去。

无饥饿：线程排队先来后到，不管优先级大小，先来先执行，就不会产生饥饿等待资源，也即公平锁；相反非公平锁则是根据优先级来执行，有可能排在前面的低优先级线程被后面的高优先级线程插队，就形成饥饿

无障碍：共享资源不加锁，每个线程都可以自有读写，单监测到被其他线程修改过则回滚操作，重试直到单独操作成功；风险就是如果多个线程发现彼此修改了，所有线程都需要回滚，就会导致死循环的回滚中，造成死锁

无锁：无锁是无障碍的加强版，无锁级别保证至少有一个线程在有限操作步骤内成功退出，不管是否修改成功，这样保证了多个线程回滚不至于导致死循环

无等待：无等待是无锁的升级版，并发编程的最高境界，无锁只保证有线程能成功退出，但存在低级别的线程一直处于饥饿状态，无等待则要求所有线程必须在有限步骤内完成退出，让低级别的线程有机会执行，从而保证所有线程都能运行，提高并发度。

#### 优势和劣势

合理使用线程是一门艺术，合理编写一道准确无误的多线程程序更是一门艺术，如果线程使用得当，能够有效的降低程序的开发和维护成本。

Java很好的在用户空间实现了开发工具包，并在内核空间提供系统调用来支持多线程编程，Java支持了丰富的类库java.util.concurrent和跨平台的内存模型，同时也提高了开发人员的门槛，并发一直以来是一个高阶的主题，但是现在，并发也成为了主流开发人员的必备素质。

虽然线程带来的好处很多，但是编写正确的多线程（并发）程序是一件极困难的事情，并发程序的 Bug 往往会诡异地出现又诡异的消失，在当你认为没有问题的时候它就出现了，难以定位 是并发程序的一个特征，所以在此基础上你需要有扎实的并发基本功。那么，并发为什么会出现呢？

#### 背景

并发为什么会出现计算机世界的快速发展离不开 CPU、内存和 I/O 设备的高速发展，但是这三者一直存在速度差异性问题，我们可以从存储器的层次结构可以看出



CPU内部是寄存器的构造，寄存器的访问速度要高于高速缓存，高速缓存的访问速度要高于内存，最慢的是磁盘访问。

程序是在内存中执行的，程序里大部分语句都要访问内存，有些还需要访问I/O设备，根据漏桶理论来说，程序整体的性能取决于最慢的操作也就是磁盘访问速度。

因为CPU速度太快了，所以为了发挥CPU的速度优势，平衡这三者的速度差异，计算机体系机构、操作系统、编译程序都做出了贡献，主要体现为：

1、CPU 使用缓存来中和和内存的访问速度差异

2、操作系统提供进程和线程调度，让 CPU在执行指令的同时分时复用线程，让内存和磁盘不断交互，不同的CPU时间片能够执行不同的任务，从而均衡这三者的差异

3、编译程序提供优化指令的执行顺序，让缓存能够合理的使用我们在享受这些便利的同时，多线程也为我们带来了挑战，下面我们就来探讨一下并发问题为什么会出现以及多线程的源头是什么？

#### 问题

##### 安全性问题

线程安全性是非常复杂的，在没有采用同步机制的情况下，多个线程中的执行操作往往是不可预测的，这也是多线程带来的挑战之一，下面我们给出一段代码，来看看安全性问题体现在哪

public class TSynchronized implements Runnable{

static int i = 0;

public void increase(){

i++;

}

@Override

public void run() {

for(int i = 0;i < 1000;i++) {

increase();

}

}

public static void main(String[] args) throws InterruptedException {

TSynchronized tSynchronized = new TSynchronized();

Thread aThread = new Thread(tSynchronized);

Thread bThread = new Thread(tSynchronized);

aThread.start();

bThread.start();

System.out.println("i = " + i);

}

}

这段程序输出后会发现，i的值每次都不一样，这不符合我们的预测，那么为什么会出现这种情况呢？我们先来分析一下程序的运行过程。TSynchronized实现了Runnable接口，并定义了一个静态变量i，然后在increase方法中每次都增加i的值，在其实现的run方法中进行循环调用，共执行1000次。

##### 可见性问题

在单核CPU时代，所有的线程共用一个CPU，CPU缓存和内存的一致性问题容易解决CPU和内存之间

如果用图来表示的话我想会是下面这样：



在多核时代，因为有多核的存在，每个核都能够独立的运行一个线程，每颗CPU都有自己的缓存，这时CPU缓存与内存的数据一致性就没那么容易解决了，当多个线程在不同的CPU上执行时，这些线程操作的是不同的CPU缓存：



因为i是静态变量，没有经过任何线程安全措施的保护，多个线程会并发修改i的值，所以我们认为i不是线程安全的，导致这种结果的出现是由于aThread和bThread中读取的 i 值彼此不可见，所以这是由于 可见性 导致的线程安全问题。

##### 原子性问题

看起来很普通的一段程序却因为两个线程aThread和bThread交替执行产生了不同的结果。但是根源不是因为创建了两个线程导致的，多线程只是产生线程安全性的必要条件，最终的根源出现在i++这个操作上。

这个操作怎么了？这不就是一个给i递增的操作吗？也就是i++ => i = i + 1，这怎么就会产生问题了？

因为i++不是一个 原子性操作，仔细想一下，i++其实有三个步骤，读取i的值，执行i + 1操作，然后把i + 1得出的值重新赋给i（将结果写入内存）。

当两个线程开始运行后，每个线程都会把i的值读入到CPU缓存中，然后执行+ 1操作，再把+ 1之后的值写入内存。因为线程间都有各自的虚拟机栈和程序计数器，他们彼此之间没有数据交换，所以当aThread 执行+ 1操作后，会把数据写入到内存，同时bThread执行+ 1操作后，也会把数据写入到内存，因为CPU时间片的执行周期是不确定的，所以会出现当aThread还没有把数据写入内存时，bThread就会读取内存中的数据，然后执行+ 1操作，再写回内存，从而覆盖i的值，导致aThread所做的努力白费。



为什么上面的线程切换会出现问题呢？

我们先来考虑一下正常情况下（即不会出现线程安全性问题的情况下）两条线程的执行顺序：



可以看到，当aThread在执行完整个i++的操作后，操作系统对线程进行切换，由 aThread -> bThread，这是最理想的操作，一旦操作系统在任意 读取/增加/写入 阶段产生线程切换，都会产生线程安全问题。例如如下图所示：



最开始的时候，内存中i = 0，aThread读取内存中的值并把它读取到自己的寄存器中，执行+1操作，此时发生线程切换，bThread开始执行，读取内存中的值并把它读取到自己的寄存器中，此时发生线程切换，线程切换至aThread开始运行，aThread把自己寄存器的值写回到内存中，此时又发生线程切换，由aThread -> bThread，线程bThread把自己寄存器的值+1然后写回内存，写完后内存中的值不是2，而是1，内存中的i值被覆盖了。

我们上面提到原子性这个概念，那么什么是原子性呢？

并发编程的原子性操作是完全独立于任何其他进程运行的操作，原子操作多用于现代操作系统和并行处理系统中。原子操作通常在内核中使用，因为内核是操作系统的主要组件。但是，大多数计算机硬件，编译器和库也提供原子性操作。在加载和存储中，计算机硬件对存储器字进行读取和写入。为了对值进行匹配、增加或者减小操作，一般通过原子操作进行。在原子操作期间，处理器可以在同一数据传输期间完成读取和写入。这样，其他输入/输出机制或处理器无法执行存储器读取或写入任务，直到原子操作完成为止。

简单来讲，就是原子操作要么全部执行，要么全部不执行。数据库事务的原子性也是基于这个概念演进的。

##### 有序性问题

在并发编程中还有带来让人非常头疼的有序性问题，有序性顾名思义就是顺序性，在计算机中指的就是指令的先后执行顺序。一个非常显而易见的例子就是 JVM 中的类加载：



这是一个JVM加载类的过程图，也称为类的生命周期，类从加载到JVM到卸载一共会经历五个阶段加载、连接、初始化、使用、卸载。这五个过程的执行顺序是一定的，但是在连接阶段，也会分为三个过程，即验证、准备、解析阶段，这三个阶段的执行顺序不是确定的，通常交叉进行，在一个阶段的执行过程中会激活另一个阶段。

有序性问题一般是编译器带来的，编译器有的时候确实是好心办坏事，它为了优化系统性能，往往更换指令的执行顺序。

##### 活跃性问题

多线程还会带来活跃性问题，如何定义活跃性问题呢？活跃性问题关注的是某件事情是否会发生。

如果一组线程中的每个线程都在等待一个事件的发生，而这个事件只能由该组中正在等待的线程触发，这种情况会导致死锁。

简单一点来表述一下，就是每个线程都在等待其他线程释放资源，而其他资源也在等待每个线程释放资源，这样没有线程抢先释放自己的资源，这种情况会产生死锁，所有线程都会无限的等待下去。

**死锁的必要条件**

造成死锁的原因有四个，破坏其中一个即可破坏死锁

互斥条件：指进程对所分配到的资源进行排它性使用，即在一段时间内某资源只由一个进程占用。如果此时还有其它进程请求资源，则请求者只能等待，直至占有资源的进程释放。

请求和保持条件：指进程已经保持至少一个资源，但又提出了新的资源请求，而该资源已被其它进程占有，此时请求进程阻塞，但又对自己已获得的其它资源保持占有。

不剥夺条件：指进程已获得的资源，在未使用完之前，不能被剥夺，只能在使用完时由自己释放。

循环等待：指在发生死锁时，必然存在一个进程对应的环形链。

换句话说，死锁线程集合中的每个线程都在等待另一个死锁线程占有的资源。但是由于所有线程都不能运行，它们之中任何一个资源都无法释放资源，所以没有一个线程可以被唤醒。

如果说死锁很痴情的话，那么活锁用一则成语来表示就是弄巧成拙。

某些情况下，当线程意识到它不能获取所需要的下一个锁时，就会尝试礼貌的释放已经获得的锁，然后等待非常短的时间再次尝试获取。可以想像一下这个场景：当两个人在狭路相逢的时候，都想给对方让路，相同的步调会导致双方都无法前进。

现在假想有一对并行的线程用到了两个资源。它们分别尝试获取另一个锁失败后，两个线程都会释放自己持有的锁，再次进行尝试，这个过程会一直进行重复。很明显，这个过程中没有线程阻塞，但是线程仍然不会向下执行，这种状况我们称之为活锁(livelock)。如果我们期望的事情一直不会发生，就会产生活跃性问题，比如单线程中的无限循环

while(true){...}

for(;;){}

在多线程中，比如aThread和bThread都需要某种资源，aThread一直占用资源不释放，bThread一直得不到执行，就会造成活跃性问题，bThread线程会产生饥饿。

##### 性能问题

与活跃性问题密切相关的是性能问题，如果说活跃性问题关注的是最终的结果，那么性能问题关注的就是造成结果的过程，性能问题有很多方面：比如服务时间过长，吞吐率过低，资源消耗过高，在多线程中这样的问题同样存在。

在多线程中，有一个非常重要的性能因素那就是我们上面提到的线程切换，也称为上下文切换(Context Switch)，这种操作开销很大。

在计算机世界中，老外都喜欢用context上下文这个词，这个词涵盖的内容很多，包括上下文切换的资源，寄存器的状态、程序计数器等。context switch一般指的就是这些上下文切换的资源、寄存器状态、程序计数器的变化等。

在上下文切换中，会保存和恢复上下文，丢失局部性，把大量的时间消耗在线程切换上而不是线程运行上。

为什么线程切换会开销如此之大呢？线程间的切换会涉及到以下几个步骤：



将CPU从一个线程切换到另一线程涉及挂起当前线程，保存其状态，例如寄存器，然后恢复到要切换的线程的状态，加载新的程序计数器，此时线程切换实际上就已经完成了；此时，CPU不在执行线程切换代码，进而执行新的和线程关联的代码。

**引起线程切换的几种方式**

线程间的切换一般是操作系统层面需要考虑的问题，那么引起线程上下文切换有哪几种方式呢？或者说线程切换有哪几种诱因呢？主要有下面几种引起上下文切换的方式：

当前正在执行的任务完成，系统的CPU正常调度下一个需要运行的线程；

当前正在执行的任务遇到 I/O 等阻塞操作，线程调度器挂起此任务，继续调度下一个任务；

多个任务并发抢占锁资源，当前任务没有获得锁资源，被线程调度器挂起，继续调度下一个任务；

用户的代码挂起当前任务，比如线程执行 sleep 方法，让出CPU；

使用硬件中断的方式引起上下文切换

##### 线程安全性

在Java中，要实现线程安全性，必须要正确的使用线程和锁，但是这些只是满足线程安全的一种方式，要编写正确无误的线程安全的代码，其核心就是对状态访问操作进行管理。最重要的就是最共享(Shared)的和可变(Mutable)的状态。只有共享和可变的变量才会出现问题，私有变量不会出现问题，参考程序计数器。

对象的状态可以理解为存储在实例变量或者静态变量中的数据，共享意味着某个变量可以被多个线程同时访问、可变意味着变量在生命周期内会发生变化。一个变量是否是线程安全的，取决于它是否被多个线程访问。要使变量能够被安全访问，必须通过同步机制来对变量进行修饰。

如果不采用同步机制的话，那么就要避免多线程对共享变量的访问，主要有下面两种方式：

不要在多线程之间共享变量

将共享变量置为不可变的

我们说了这么多次线程安全性，那么什么是线程安全性呢？

**什么是线程安全性**

多个线程可以同时安全调用的代码称为线程安全的，如果一段代码是安全的，那么这段代码就不存在竞态条件。仅仅当多个线程共享资源时，才会出现竞态条件。

根据上面的探讨，我们可以得出一个简单的结论：当多个线程访问某个类时，这个类始终都能表现出正确的行为，那么就称这个类是线程安全的。

单线程就是一个线程数量为1的多线程，单线程一定是线程安全的。读取某个变量的值不会产生安全性问题，因为不管读取多少次，这个变量的值都不会被修改。

###### 原子性

我们上面提到了原子性的概念，你可以把原子性操作想象成为一个不可分割 的整体，它的结果只有两种，要么全部执行，要么全部回滚。你可以把原子性认为是婚姻关系的一种，男人和女人只会产生两种结果，好好的和说散就散，一般男人的一生都可以把他看成是原子性的一种，当然我们不排除时间管理(线程切换)的个例，我们知道线程切换必然会伴随着安全性问题，男人要出去浪也会造成两种结果，这两种结果分别对应安全性的两个结果：线程安全（好好的）和线程不安全（说散就散）。

###### 竞态条件

有了上面的线程切换的功底，那么竞态条件也就好定义了，它指的就是两个或多个线程同时对一共享数据进行修改，从而影响程序运行的正确性时，这种就被称为竞态条件(race condition) ，线程切换是导致竞态条件出现的诱导因素，我们通过一个示例来说明，来看一段代码：

public class RaceCondition {

private Signleton single = null;

public Signleton newSingleton(){

if(single == null){

single = new Signleton();

}

return single;

}

}

在上面的代码中，涉及到一个竞态条件，那就是判断single的时候，如果single判断为空，此时发生了线程切换，另外一个线程执行，判断single的时候，也是空，执行new操作，然后线程切换回之前的线程，再执行new操作，那么内存中就会有两个Singleton对象。

###### 加锁机制

在Java中，有很多种方式来对共享和可变的资源进行加锁和保护。Java提供一种内置的机制对资源进行保护：synchronized关键字，它有三种保护机制：

对方法进行加锁，确保多个线程中只有一个线程执行方法；

对某个对象实例（在我们上面的探讨中，变量可以使用对象来替换）进行加锁，确保多个线程中只有一个线程对对象实例进行访问；

对类对象进行加锁，确保多个线程只有一个线程能够访问类中的资源。

synchronized 关键字对资源进行保护的代码块俗称同步代码块(Synchronized Block)，例如

synchronized(lock){

// 线程安全的代码

}

每个Java对象都可以用做一个实现同步的锁，这些锁被称为内置锁(Instrinsic Lock)或者监视器锁(Monitor Lock)。线程在进入同步代码之前会自动获得锁，并且在退出同步代码时自动释放锁，而无论是通过正常执行路径退出还是通过异常路径退出，获得内置锁的唯一途径就是进入这个由锁保护的同步代码块或方法。

synchronized的另一种隐含的语义就是互斥，互斥意味着独占，最多只有一个线程持有锁，当线程A尝试获得一个由线程B持有的锁时，线程A必须等待或者阻塞，直到线程B释放这个锁，如果线程B不释放锁的话，那么线程A将会一直等待下去。线程A获得线程B持有的锁时，线程A必须等待或者阻塞，但是获取锁的线程B可以重入，重入的意思可以用一段代码表示

public class Retreent {

public synchronized void doSomething(){

doSomethingElse();

System.out.println("doSomething......");

}

public synchronized void doSomethingElse(){

System.out.println("doSomethingElse......");

}

获取 doSomething()方法锁的线程可以执行doSomethingElse()方法，执行完毕后可以重新执行 doSomething()方法中的内容。锁重入也支持子类和父类之间的重入。

volatile是一种轻量级的synchronized，也就是一种轻量级的加锁方式，volatile通过保证共享变量的可见性来从侧面对对象进行加锁。可见性的意思就是当一个线程修改一个共享变量时，另外一个线程能够 看见 这个修改的值。volatile的执行成本要比synchronized低很多，因为volatile不会引起线程的上下文切换。



我们还可以使用原子类 来保证线程安全，原子类其实就是rt.jar下面以atomic开头的类：



除此之外，我们还可以使用 java.util.concurrent工具包下的线程安全的集合类来确保线程安全，具体的实现类和其原理我们后面会说。可以使用不同的并发模型来实现并发系统，并发模型说的是系统中的线程如何协作完成并发任务。不同的并发模型以不同的方式拆分任务，线程可以以不同的方式进行通信和协作。

###### 竞态条件和关键区域

竞态条件是在关键代码区域发生的一种特殊条件。关键区域是由多个线程同时执行的代码部分，关键区域中的代码执行顺序会对造成不一样的结果。如果多个线程执行一段关键代码，而这段关键代码会因为执行顺序不同而造成不同的结果时，那么这段代码就会包含竞争条件。

#### 线程单元

##### 状态

###### 线程状态

###### 生命周期

##### 动作

#### 线程交互

#### 线程工具

##### 同步控制

##### 线程池

##### 并发容器

#### 线程调优

##### 性能指标

##### 锁优化

##### JVM锁机制

##### 无锁

#### 并发模型

**并发模型和分布式系统很相似**

并发模型其实和分布式系统模型非常相似，在并发模型中是线程彼此进行通信，而在分布式系统模型中是 进程 彼此进行通信。然而本质上，进程和线程也非常相似。这也就是为什么并发模型和分布式模型非常相似的原因。

分布式系统通常要比并发系统面临更多的挑战和问题比如进程通信、网络可能出现异常，或者远程机器挂掉等等。但是一个并发模型同样面临着比如 CPU 故障、网卡出现问题、硬盘出现问题等。

因为并发模型和分布式模型很相似，因此他们可以相互借鉴，例如用于线程分配的模型就类似于分布式系统环境中的负载均衡模型。

其实说白了，分布式模型的思想就是借鉴并发模型的基础上推演发展来的。

**认识两个状态**

并发模型的一个重要的方面是，线程是否应该共享状态，是具有共享状态还是独立状态。共享状态也就意味着在不同线程之间共享某些状态状态。

其实就是数据，比如一个或者多个对象。当线程要共享数据时，就会造成竞态条件或者死锁等问题。当然，这些问题只是可能会出现，具体实现方式取决于你是否安全的使用和访问共享对象。



独立的状态表明状态不会在多个线程之间共享，如果线程之间需要通信的话，他们可以访问不可变的对象来实现，这是最有效的避免并发问题的一种方式，如下图所示



使用独立状态让我们的设计更加简单，因为只有一个线程能够访问对象，即使交换对象，也是不可变的对象。

**并发模型：**

##### 并行Worker

###### 概述

第一个并发模型是并行worker模型，客户端会把任务交给代理人(Delegator)，然后由代理人把工作分配给不同的工人(worker)。如下图所示



并行worker的核心思想是，它主要有两个进程即代理人和工人，Delegator负责接收来自客户端的任务并把任务下发，交给具体的Worker进行处理，Worker处理完成后把结果返回给Delegator，在Delegator接收到Worker处理的结果后对其进行汇总，然后交给客户端。并行 Worker 模型是 Java 并发模型中非常常见的一种模型。许多 java.util.concurrent 包下的并发工具都使用了这种模型。

###### 优点

并行 Worker 模型的一个非常明显的特点就是很容易理解，为了提高系统的并行度你可以增加多个 Worker 完成任务。

并行 Worker 模型的另外一个好处就是，它会将一个任务拆分成多个小任务，并发执行，Delegator 在接受到 Worker 的处理结果后就会返回给 Client，整个 Worker -> Delegator -> Client 的过程是异步的。

###### 缺点

并行 Worker 模式同样会有一些隐藏的缺点：

**共享状态会变得很复杂**

实际的并行 Worker 要比我们图中画出的更复杂，主要是并行 Worker 通常会访问内存或共享数据库中的某些共享数据。



这些共享状态可能会使用一些工作队列来保存业务数据、数据缓存、数据库的连接池等。在线程通信中，线程需要确保共享状态是否能够让其他线程共享，而不是仅仅停留在CPU 缓存中让自己可用，当然这些都是程序员在设计时就需要考虑的问题。线程需要避免竞态条件，死锁和许多其他共享状态造成的并发问题。

多线程在访问共享数据时，会丢失并发性，因为操作系统要保证只有一个线程能够访问数据，这会导致共享数据的争用和抢占。未抢占到资源的线程会阻塞。

现代的非阻塞并发算法可以减少争用提高性能，但是非阻塞算法比较难以实现。

可持久化的数据结构(Persistent data structures) 是另外一个选择。可持久化的数据结构在修改后始终会保留先前版本。因此，如果多个线程同时修改一个可持久化的数据结构，并且一个线程对其进行了修改，则修改的线程会获得对新数据结构的引用。

虽然可持久化的数据结构是一个新的解决方法，但是这种方法实行起来却有一些问题，比如，一个持久列表会将新元素添加到列表的开头，并返回所添加的新元素的引用，但是其他线程仍然只持有列表中先前的第一个元素的引用，他们看不到新添加的元素。

持久化的数据结构比如链表(LinkedList) 在硬件性能上表现不佳。列表中的每个元素都是一个对象，这些对象散布在计算机内存中。现代CPU的顺序访问往往要快的多，因此使用数组等顺序访问的数据结构则能够获得更高的性能。CPU高速缓存可以将一个大的矩阵块加载到高速缓存中，并让CPU在加载后直接访问CPU高速缓存中的数据。对于链表，将元素分散在整个 RAM 上，这实际上是不可能的。

**无状态的worker**

共享状态可以由其他线程所修改，因此，worker 必须在每次操作共享状态时重新读取，以确保在副本上能够正确工作。不在线程内部保持状态的 worker 成为无状态的 worker。

**作业顺序是不确定的**

并行工作模型的另一个缺点是作业的顺序不确定，无法保证首先执行或最后执行哪些作业。任务 A 在任务 B 之前分配给 worker，但是任务 B 可能在任务 A 之前执行。

##### 流水线

###### 概述

第二种并发模型就是我们经常在生产车间遇到的流水线并发模型，下面是流水线设计模型的流程图：



这种组织架构就像是工厂中装配线中的worker，每个worker只完成全部工作的一部分，完成一部分后，worker会将工作转发给下一个worker。

每道程序都在自己的线程中运行，彼此之间不会共享状态，这种模型也被称为无共享并发模型。

使用流水线并发模型通常被设计为非阻塞I/O，也就是说，当没有给worker分配任务时，worker会做其他工作。非阻塞I/O意味着当worker开始I/O操作，例如从网络中读取文件，worker不会等待I/O调用完成。因为I/O操作很慢，所以等待I/O非常耗费时间。在等待 I/O的同时，CPU可以做其他事情，I/O操作完成后的结果将传递给下一个worker。下面是非阻塞I/O的流程图：



在实际情况中，任务通常不会按着一条装配线流动，由于大多数程序需要做很多事情，因此需要根据完成的不同工作在不同的worker之间流动，如下图所示：



任务还可能需要多个worker共同参与完成：



###### 优点

与并行设计模型相比，流水线模型具有一些优势，具体优势如下：

**不会存在共享状态**

因为流水线设计能够保证worker在处理完成后再传递给下一个worker，所以worker与worker之间不需要共享任何状态，也就无需考虑并发问题。你甚至可以在实现上把每个worker看成是单线程的一种。

**有状态worker**

因为worker知道没有其他线程修改自身的数据，所以流水线设计中的worker是有状态的，有状态的意思是他们可以将需要操作的数据保留在内存中，有状态通常比无状态更快。

**更好的硬件**

整合因为你可以把流水线看成是单线程的，而单线程的工作优势在于它能够和硬件的工作方式相同。因为有状态的worker通常在 CPU 中缓存数据，这样可以更快地访问缓存的数据。

**使任务更加有效的进行**

可以对流水线并发模型中的任务进行排序，一般用来日志的写入和恢复。

###### 缺点

流水线并发模型的缺点是任务会涉及多个worker，因此可能会分散在项目代码的多个类中。因此很难确定每个worker都在执行哪个任务。流水线的代码编写也比较困难，设计许多嵌套回调处理程序的代码通常被称为 回调地狱。回调地狱很难追踪debug。

##### 响应式-事件驱动系统

使用流水线模型的系统有时也被称为响应式或者事件驱动系统，这种模型会根据外部的事件作出响应，事件可能是某个HTTP请求或者某个文件完成加载到内存中。

###### Actor模型

在 Actor 模型中，每一个 Actor 其实就是一个 Worker， 每一个 Actor 都能够处理任务。简单来说，Actor 模型是一个并发模型，它定义了一系列系统组件应该如何动作和交互的通用规则，最著名的使用这套规则的编程语言是 Erlang。一个参与者Actor对接收到的消息做出响应，然后可以创建出更多的 Actor 或发送更多的消息，同时准备接收下一条消息。



###### Channels模型

在Channel模型中，worker 通常不会直接通信，与此相对的，他们通常将事件发送到不同的通道(Channel)上，然后其他worker可以在这些通道上获取消息，下面是Channel的模型图：



有的时候worker不需要明确知道接下来的worker是谁，他们只需要将作者写入通道中，监听 Channel 的 worker 可以订阅或者取消订阅，这种方式降低了 worker 和 worker 之间的耦合性。

##### 函数性并行

函数性并行模型是最近才提出的一种并发模型，它的基本思路是使用函数调用来实现。消息的传递就相当于是函数的调用。传递给函数的参数都会被拷贝，因此在函数之外的任何实体都无法操纵函数内的数据。这使得函数执行类似于原子操作。

每个函数调用都可以独立于任何其他函数调用执行。当每个函数调用独立执行时，每个函数都可以在单独的 CPU 上执行。这也就是说，函数式并行并行相当于是各个 CPU 单独执行各自的任务。

JDK 1.7 中的 ForkAndJoinPool 类就实现了函数性并行的功能。Java 8 提出了 stream 的概念，使用并行流也能够实现大量集合的迭代。

函数性并行的难点是要知道函数的调用流程以及哪些 CPU 执行了哪些函数，跨 CPU 函数调用会带来额外的开销。

我们之前说过，线程就是进程中的一条顺序流，在 Java 中，每一条 Java 线程就像是 JVM 的一条顺序流，就像是虚拟 CPU 一样来执行代码。Java 中的 main() 方法是一条特殊的线程，JVM 创建的 main 线程是一条主执行线程，在 Java 中，方法都是由 main 方法发起的。在 main 方法中，你照样可以创建其他的线程(执行顺序流)，这些线程可以和 main 方法共同执行应用代码。

Java 线程也是一种对象，它和其他对象一样。Java 中的 Thread 表示线程，Thread 是 java.lang.Thread 类或其子类的实例。那么下面我们就来一起探讨一下在 Java 中如何创建和启动线程。

###### 创建并启动线程

在 Java 中，创建线程的方式主要有三种：

通过继承 Thread 类来创建线程

通过实现 Runnable 接口来创建线程

通过 Callable 和 Future 来创建线程

下面我们分别探讨一下这几种创建方式：

继承Thread类来创建线程

第一种方式是继承 Thread 类来创建线程，如下示例

public class TJavaThread extends Thread{

static int count;

@Override

public synchronized void run() {

for(int i = 0;i < 10000;i++){

count++;

}

}

public static void main(String[] args) throws InterruptedException {

TJavaThread tJavaThread = new TJavaThread();

tJavaThread.start();

tJavaThread.join();

System.out.println("count = " + count);

}

}

线程的主要创建步骤如下：

定义一个线程类使其继承 Thread 类，并重写其中的 run 方法，run 方法内部就是线程要完成的任务，因此 run 方法也被称为执行体

创建了 Thread 的子类，上面代码中的子类是 TJavaThread

启动方法需要注意，并不是直接调用 run 方法来启动线程，而是使用 start 方法来启动线程。当然 run 方法可以调用，这样的话就会变成普通方法调用，而不是新创建一个线程来调用了。

public static void main(String[] args) throws InterruptedException {

TJavaThread tJavaThread = new TJavaThread();

tJavaThread.run();

System.out.println("count = " + count);

}

这样的话，整个 main 方法只有一条执行线程也就是 main 线程，由两条执行线程变为一条执行线程



Thread 构造器只需要一个 Runnable 对象，调用 Thread 对象的 start() 方法为该线程执行必须的初始化操作，然后调用 Runnable 的 run 方法，以便在这个线程中启动任务。我们上面使用了线程的 join 方法，它用来等待线程的执行结束，如果我们不加 join 方法，它就不会等待 tJavaThread 的执行完毕，输出的结果可能就不是 10000



可以看到，在 run 方法还没有结束前，run 就被返回了。也就是说，程序不会等到 run 方法执行完毕就会执行下面的指令。

使用继承方式创建线程的优势：编写比较简单；可以使用 this 关键字直接指向当前线程，而无需使用 Thread.currentThread() 来获取当前线程。

使用继承方式创建线程的劣势：在 Java 中，只允许单继承（拒绝肛精说使用内部类可以实现多继承）的原则，所以使用继承的方式，子类就不能再继承其他类。

使用Runnable接口来创建线程

相对的，还可以使用Runnable接口来创建线程，如下示例

public class TJavaThreadUseImplements implements Runnable{

static int count;

@Override

public synchronized void run() {

for(int i = 0;i < 10000;i++){

count++;

}

}

public static void main(String[] args) throws InterruptedException {

new Thread(new TJavaThreadUseImplements()).start();

System.out.println("count = " + count);

}

}

线程的主要创建步骤如下

1、首先定义 Runnable 接口，并重写 Runnable 接口的 run 方法，run 方法的方法体同样是该线程的线程执行体。

2、创建线程实例，可以使用上面代码这种简单的方式创建，也可以通过 new 出线程的实例来创建，如下所示

TJavaThreadUseImplements tJavaThreadUseImplements = new TJavaThreadUseImplements();

new Thread(tJavaThreadUseImplements).start();

3、再调用线程对象的 start 方法来启动该线程。

线程在使用实现 Runnable 的同时也能实现其他接口，非常适合多个相同线程来处理同一份资源的情况，体现了面向对象的思想。

使用Runnable实现的劣势是编程稍微繁琐，如果要访问当前线程，则必须使用 Thread.currentThread() 方法。

使用Callable接口来创建线程

Runnable 接口执行的是独立的任务，Runnable 接口不会产生任何返回值，如果你希望在任务完成后能够返回一个值的话，那么你可以实现 Callable 接口而不是 Runnable 接口。Java SE5 引入了 Callable 接口，它的示例如下

public class CallableTask implements Callable {

static int count;

public CallableTask(int count){

this.count = count;

}

@Override

public Object call() {

return count;

}

public static void main(String[] args) throws ExecutionException, InterruptedException {

FutureTask<Integer> task = new FutureTask((Callable<Integer>) () -> {

for(int i = 0;i < 1000;i++){

count++;

}

return count;

});

Thread thread = new Thread(task);

thread.start();

Integer total = task.get();

System.out.println("total = " + total);

}

}

我想，使用 Callable 接口的好处你已经知道了吧，既能够实现多个接口，也能够得到执行结果的返回值。Callable 和 Runnable 接口还是有一些区别的，主要区别如下

Callable 执行的任务有返回值，而 Runnable 执行的任务没有返回值

Callable（重写）的方法是 call 方法，而 Runnable（重写）的方法是 run 方法。

call 方法可以抛出异常，而 Runnable 方法不能抛出异常

使用线程池来创建线程

首先先来认识一下顶级接口 Executor，Executor 虽然不是传统线程创建的方式之一，但是它却成为了创建线程的替代者，使用线程池的好处如下

利用线程池能够复用线程、控制最大并发数。

实现任务线程队列缓存策略和拒绝机制。

实现某些与时间相关的功能，如定时执行、周期执行等。

隔离线程环境。比如，交易服务和搜索服务在同一台服务器上，分别开启两个线程池，交易线程的资源消耗明显要大；因此，通过配置独立的线程池，将较慢的交易服务与搜索服务隔开，避免个服务线程互相影响。

你可以使用如下操作来替换线程创建

new Thread(new(RunnableTask())).start()

// 替换为

Executor executor = new ExecutorSubClass() // 线程池实现类;

executor.execute(new RunnableTask1());

executor.execute(new RunnableTask2());

ExecutorService是Executor的默认实现，也是Executor 的扩展接口，ThreadPoolExecutor 类提供了线程池的扩展实现。Executors类为这些Executor提供了方便的工厂方法。下面是使用ExecutorService创建线程的几种方式

CachedThreadPool

从而简化了并发编程。Executor 在客户端和任务之间提供了一个间接层；与客户端直接执行任务不同，这个中介对象将执行任务。Executor 允许你管理异步任务的执行，而无须显示地管理线程的生命周期。

public static void main(String[] args) {

ExecutorService service = Executors.newCachedThreadPool();

for(int i = 0;i < 5;i++){

service.execute(new TestThread());

}

service.shutdown();

}

CachedThreadPool 会为每个任务都创建一个线程。注意：ExecutorService 对象是使用静态的 Executors 创建的，这个方法可以确定 Executor 类型。对 shutDown 的调用可以防止新任务提交给 ExecutorService ，这个线程在 Executor 中所有任务完成后退出。

FixedThreadPoolFixedThreadPool

使你可以使用有限的线程集来启动多线程

public static void main(String[] args) {

ExecutorService service = Executors.newFixedThreadPool(5);

for(int i = 0;i < 5;i++){

service.execute(new TestThread());

}

service.shutdown();

}

有了 FixedThreadPool 使你可以一次性的预先执行高昂的线程分配，因此也就可以限制线程的数量。这可以节省时间，因为你不必为每个任务都固定的付出创建线程的开销。

SingleThreadExecutor

SingleThreadExecutor 就是线程数量为 1的 FixedThreadPool，如果向 SingleThreadPool 一次性提交了多个任务，那么这些任务将会排队，每个任务都会在下一个任务开始前结束，所有的任务都将使用相同的线程。SingleThreadPool 会序列化所有提交给他的任务，并会维护它自己(隐藏)的悬挂队列。

public static void main(String[] args) {

ExecutorService service = Executors.newSingleThreadExecutor();

for(int i = 0;i < 5;i++){

service.execute(new TestThread());

}

service.shutdown();

}

从输出的结果就可以看到，任务都是挨着执行的。我为任务分配了五个线程，但是这五个线程不像是我们之前看到的有换进换出的效果，它每次都会先执行完自己的那个线程，然后余下的线程继续走完这条线程的执行路径。你可以用 SingleThreadExecutor 来确保任意时刻都只有唯一一个任务在运行。

休眠

影响任务行为的一种简单方式就是使线程 休眠，选定给定的休眠时间，调用它的 sleep() 方法，一般使用的TimeUnit 这个时间类替换 Thread.sleep() 方法，示例如下：

public class SuperclassThread extends TestThread{

@Override

public void run() {

System.out.println(Thread.currentThread() + "starting ..." );

try {

for(int i = 0;i < 5;i++){

if(i == 3){

System.out.println(Thread.currentThread() + "sleeping ...");

TimeUnit.MILLISECONDS.sleep(1000);

}

}

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread() + "wakeup and end ...");

}

public static void main(String[] args) {

ExecutorService executors = Executors.newCachedThreadPool();

for(int i = 0;i < 5;i++){

executors.execute(new SuperclassThread());

}

executors.shutdown();

}

}

关于 TimeUnit 中的 sleep() 方法和 Thread.sleep() 方法的比较，请参考下面这篇博客(https://www.cnblogs.com/xiadongqing/p/9925567.html)

优先级

上面提到线程调度器对每个线程的执行都是不可预知的，随机执行的，那么有没有办法告诉线程调度器哪个任务想要优先被执行呢？你可以通过设置线程的优先级状态，告诉线程调度器哪个线程的执行优先级比较高，请给这个骑手马上派单，线程调度器倾向于让优先级较高的线程优先执行，然而，这并不意味着优先级低的线程得不到执行，也就是说，优先级不会导致死锁的问题。优先级较低的线程只是执行频率较低。

public class SimplePriorities implements Runnable{

private int priority;

public SimplePriorities(int priority) {

this.priority = priority;

}

@Override

public void run() {

Thread.currentThread().setPriority(priority);

for(int i = 0;i < 100;i++){

System.out.println(this);

if(i % 10 == 0){

Thread.yield();

}

}

}

@Override

public String toString() {

return Thread.currentThread() + " " + priority;

}

public static void main(String[] args) {

ExecutorService service = Executors.newCachedThreadPool();

for(int i = 0;i < 5;i++){

service.execute(new SimplePriorities(Thread.MAX\_PRIORITY));

}

service.execute(new SimplePriorities(Thread.MIN\_PRIORITY));

}

}

toString()方法被覆盖，以便通过使用Thread.toString()方法来打印线程的名称。你可以改写线程的默认输出，这里采用了Thread[pool-1-thread-1,10,main]这种形式的输出。

通过输出，你可以看到，最后一个线程的优先级最低，其余的线程优先级最高。注意，优先级是在run开头设置的，在构造器中设置它们不会有任何好处，因为这个时候线程还没有执行任务。

尽管JDK有10个优先级，但是一般只有MAX\_PRIORITY，NORM\_PRIORITY，MIN\_PRIORITY 三种级别。

作出让步

我们上面提过，如果知道一个线程已经在 run() 方法中运行的差不多了，那么它就可以给线程调度器一个提示：我已经完成了任务中最重要的部分，可以让给别的线程使用 CPU 了。这个暗示将通过 yield() 方法作出。有一个很重要的点就是，Thread.yield() 是建议执行切换CPU，而不是强制执行CPU切换。对于任何重要的控制或者在调用应用时，都不能依赖于 yield()方法，实际上， yield() 方法经常被滥用。

后台线程

后台(daemon)线程，是指运行时在后台提供的一种服务线程，这种线程不是属于必须的。当所有非后台线程结束时，程序也就停止了，同时会终止所有的后台线程。反过来说，只要有任何非后台线程还在运行，程序就不会终止。

public class SimpleDaemons implements Runnable{

@Override

public void run() {

while (true){

try {

TimeUnit.MILLISECONDS.sleep(100);

System.out.println(Thread.currentThread() + " " + this);

} catch (InterruptedException e) {

System.out.println("sleep() interrupted");

}

}

}

public static void main(String[] args) throws InterruptedException {

for(int i = 0;i < 10;i++){

Thread daemon = new Thread(new SimpleDaemons());

daemon.setDaemon(true);

daemon.start();

}

System.out.println("All Daemons started");

TimeUnit.MILLISECONDS.sleep(175);

}

}

在每次的循环中会创建 10 个线程，并把每个线程设置为后台线程，然后开始运行，for 循环会进行十次，然后输出信息，随后主线程睡眠一段时间后停止运行。在每次 run 循环中，都会打印当前线程的信息，主线程运行完毕，程序就执行完毕了。因为 daemon 是后台线程，无法影响主线程的执行。但是当你把 daemon.setDaemon(true) 去掉时，while(true) 会进行无限循环，那么主线程一直在执行最重要的任务，所以会一直循环下去无法停止。

ThreadFactory

按需要创建线程的对象。使用线程工厂替换了 Thread 或者 Runnable 接口的硬连接，使程序能够使用特殊的线程子类，优先级等。一般的创建方式为

class SimpleThreadFactory implements ThreadFactory {

public Thread newThread(Runnable r) {

return new Thread(r);

}

}

Executors.defaultThreadFactory 方法提供了一个更有用的简单实现，它在返回之前将创建的线程上下文设置为已知值ThreadFactory是一个接口，它只有一个方法就是创建线程的方法

public interface ThreadFactory {

//构建一个新的线程。实现类可能初始化优先级，名称，后台线程状态和 线程组等

Thread newThread(Runnable r);

}

下面来看一个 ThreadFactory 的例子public class DaemonThreadFactory implements ThreadFactory {

@Override

public Thread newThread(Runnable r) {

Thread t = new Thread(r);

t.setDaemon(true);

return t;

}

}

public class DaemonFromFactory implements Runnable{

@Override

public void run() {

while (true){

try {

TimeUnit.MILLISECONDS.sleep(100);

System.out.println(Thread.currentThread() + " " + this);

} catch (InterruptedException e) {

System.out.println("Interrupted");

}

}

}

public static void main(String[] args) throws InterruptedException {

ExecutorService service = Executors.newCachedThreadPool(new DaemonThreadFactory());

for(int i = 0;i < 10;i++){

service.execute(new DaemonFromFactory());

}

System.out.println("All daemons started");

TimeUnit.MILLISECONDS.sleep(500);

}

}

Executors.newCachedThreadPool 可以接受一个线程池对象，创建一个根据需要创建新线程的线程池，但会在它们可用时重用先前构造的线程，并在需要时使用提供的 ThreadFactory 创建新线程。

public static ExecutorService newCachedThreadPool(ThreadFactory threadFactory) {

return new ThreadPoolExecutor(0, Integer.MAX\_VALUE,

60L, TimeUnit.SECONDS,

new SynchronousQueue<Runnable>(),

threadFactory);

}

加入一个线程一个线程可以在其他线程上调用 join() 方法，其效果是等待一段时间直到第二个线程结束才正常执行。如果某个线程在另一个线程 t 上调用 t.join() 方法，此线程将被挂起，直到目标线程 t 结束才回复(可以用 t.isAlive() 返回为真假判断)。也可以在调用 join 时带上一个超时参数，来设置到期时间，时间到期，join方法自动返回。对 join 的调用也可以被中断，做法是在线程上调用 interrupted 方法，这时需要用到 try...catch 子句

public class TestJoinMethod extends Thread{

@Override

public void run() {

for(int i = 0;i < 5;i++){

try {

TimeUnit.MILLISECONDS.sleep(1000);

} catch (InterruptedException e) {

System.out.println("Interrupted sleep");

}

System.out.println(Thread.currentThread() + " " + i);

}

}

public static void main(String[] args) throws InterruptedException {

TestJoinMethod join1 = new TestJoinMethod();

TestJoinMethod join2 = new TestJoinMethod();

TestJoinMethod join3 = new TestJoinMethod();

join1.start();

// join1.join();

join2.start();

join3.start();

}

}

join() 方法等待线程死亡。 换句话说，它会导致当前运行的线程停止执行，直到它加入的线程完成其任务。线程异常捕获由于线程的本质，使你不能捕获从线程中逃逸的异常，一旦异常逃出任务的 run 方法，它就会向外传播到控制台，除非你采取特殊的步骤捕获这种错误的异常，在 Java5 之前，你可以通过线程组来捕获，但是在 Java 5 之后，就需要用 Executor 来解决问题，因为线程组不是一次好的尝试。下面的任务会在 run 方法的执行期间抛出一个异常，并且这个异常会抛到 run 方法的外面，而且 main 方法无法对它进行捕获

public class ExceptionThread implements Runnable{

@Override

public void run() {

throw new RuntimeException();

}

public static void main(String[] args) {

try {

ExecutorService service = Executors.newCachedThreadPool();

service.execute(new ExceptionThread());

}catch (Exception e){

System.out.println("eeeee");

}

}

}为了解决这个问题，我们需要修改 Executor 产生线程的方式，Java5 提供了一个新的接口 Thread.UncaughtExceptionHandler ，它允许你在每个 Thread 上都附着一个异常处理器。Thread.UncaughtExceptionHandler.uncaughtException() 会在线程因未捕获临近死亡时被调用。

public class ExceptionThread2 implements Runnable{

@Override

public void run() {

Thread t = Thread.currentThread();

System.out.println("run() by " + t);

System.out.println("eh = " + t.getUncaughtExceptionHandler());

// 手动抛出异常

throw new RuntimeException();

}

}

// 实现Thread.UncaughtExceptionHandler 接口，创建异常处理器

public class MyUncaughtExceptionHandler implements Thread.UncaughtExceptionHandler{

@Override

public void uncaughtException(Thread t, Throwable e) {

System.out.println("caught " + e);

}

}

public class HandlerThreadFactory implements ThreadFactory {

@Override

public Thread newThread(Runnable r) {

System.out.println(this + " creating new Thread");

Thread t = new Thread(r);

System.out.println("created " + t);

t.setUncaughtExceptionHandler(new MyUncaughtExceptionHandler());

System.out.println("ex = " + t.getUncaughtExceptionHandler());

return t;

}

}

public class CaptureUncaughtException {

public static void main(String[] args) {

ExecutorService service = Executors.newCachedThreadPool(new HandlerThreadFactory());

service.execute(new ExceptionThread2());

}

}

在程序中添加了额外的追踪机制，用来验证工厂创建的线程会传递给UncaughtExceptionHandler，你可以看到，未捕获的异常是通过 uncaughtException 来捕获的。

### 线程执行

如果有两个任务需要处理，一个任务A，一个任务B

方案一：一个线程执行任务A和B，A执行完后，执行B

方案二：两个线程A和B去执行任务A 和 B，同时进行

哪个方案更快？

线程的执行，是由CPU进行调度的，一个CPU在同一时刻只会执行一个线程，我们看上去的线程A和线程B并发执行。

为了让用户感觉这些任务正在同时进行，操作系统利用了时间片轮转的方式，CPU给每个任务都服务一定的时间，然后把当前任务的状态保存下来，在加载下一任务的状态后，继续服务下一任务。任务的状态保存及再加载，这段过程就叫做上下文切换。

上下文切换过程是需要时间的；现在我们来看一下上面的问题，小伙伴们再看一下是哪个方案快呢？是不是有人会说方案一，因为不需要线程切换；方案二需要来回切换这两个线程，耗时会多点。

### 提升QPS/TPS

衡量系统性能如何，主要指标系统的（QPS/TPS）

QPS/TPS：每秒能够处理请求/事务的数量

并发数：系统同时处理的请求/事务的数量

响应时间：就是平均处理一个请求/事务需要时长

QPS/TPS = 并发数/响应时间

上面公式代表并发数越大，QPS就越大；所以很多人就会以为调大线程池，并发数就会大，也会提升QPS，所以才会出现一开始前言所说的，大多数人的误区。

其实QPS还跟响应时间成反比，响应时间越大，QPS就会越小。

虽然并发数调大了，就会提升QPS，但线程数也会影响响应时间，因为上面我们也提到了上下文切换的问题，那怎么设置线程数的呢？

#### 设置线程数

那我们如何分配线程？我们提供一个公式：

**最佳线程数目 = （（线程等待时间+线程CPU时间）/线程CPU时间 ）\* CPU数目**

备注这个公式也是前辈们分享的，当然之前看了淘宝前台系统优化实践的文章，和上面的公式很类似，不过在CPU数目那边，他们更细化了，上面的公式只是参考。不过不管什么公式，最终还是在生产环境中运行后，再优化调整。

我们继续上面的任务，我们的服务器CPU核数为4核，一个任务线程cpu耗时为20ms，线程等待（网络IO、磁盘IO）耗时80ms，那最佳线程数目：( 80 + 20 )/20 \* 4 = 20。也就是设置20个线程数最佳。

从这个公式上面我们就得出，线程的等待时间越大，线程数就要设置越大，这个正好符合我们上面的分析，可提升CPU利用率。那从另一个角度上面说，线程数设置多大，是根据我们自身的业务的，需要自己去压力测试，设置一个合理的数值。

#### 基础常规标准

因为很多业务集中到一个线程池中，不像上面的案例比较简单，事实上业务太多，怎么设置呢？这个就是要去压力测试去调整。不过我们的前辈已经帮我们总结了一个基础的值（最终还是要看运行情况自行调整）

1、CPU密集型：操作内存处理的业务，一般线程数设置为：**CPU核数+ 1或者CPU核数\*2**。核数为4的话，一般设置5或8

2、IO密集型：文件操作，网络操作，数据库操作，一般线程设置为：**cpu核数 / (1-0.9)**，核数为4的话，一般设置40

#### 绑核

**为什么要绑核？**

充分利用CPU，减少CPU之间上下文切换

指定程序运行在指定CPU，便于区分

$ taskset -c 1 ./proName

将proName绑定在第二个核。

$ taskset -c 1-3 ./proName

绑定运行在第二个到第四个核。

$ taskset -p 3569

pid 3569's current affinity mask: f

查看进程3569当前运行在哪个核上。

mask f转为二进制即为1111，因此四个核都有运行。

当然除了命令行，还有函数接口可以使用。

### 多线程交替打印

参考：

<https://blog.csdn.net/wkb15002993591/article/details/107046097/>

# 联系/区别

## 联系

通常在一个进程中可以包含若干个线程，它们可以利用进程所拥有的资源。但是，一个线程只属于一个进程。

进程间相互独立，同一进程的各线程间共享。某进程内的线程在其它进程不可见。而且需要注意的是，线程不是一个可执行的实体。

## 区别

**进程是cpu资源分配的最小单位，线程是cpu调度的最小单位。**

进行和线程之间的差异可以从下面几个方面来阐述：

### 调度

**调度：**在引入线程的操作系统中，**线程是调度和分配的基本单位，进程是资源拥有的基本单位**。把传统进程的两个属性分开，线程便能轻装运行，从而可显著地提高系统的并发程度。在同一进程中，线程的切换不会引起进程的切换；在由一个进程中的线程切换到另一个进程中的线程时，才会引起进程的切换。

### 并发性

**并发性：**在引入线程的操作系统中，不仅进程之间可以并发执行，而且在一个进程中的多个线程之间亦可并发执行，因而使操作系统具有更好的并发性，从而能更有效地使用系统资源和提高系统吞吐量。

### 拥有资源

**拥有资源：**不论是传统的操作系统，还是设有线程的操作系统，进程都是拥有资源的一个独立单位，它可以拥有自己的资源。一般地说，线程自己不拥有系统资源（只有一些必不可少的资源），但它可以访问其隶属进程的资源。

### 系统开销

**系统开销：**由于在创建或撤消进程时，系统都要为之分配或回收资源，因此，操作系统所付出的开销将显著地大于在创建或撤消线程时的开销。进程切换的开销也远大于线程切换的开销。

### 通信

**通信：**进程间通信IPC，线程间可以直接读写进程数据段（如全局变量）来进行通信——需要进程同步和互斥手段的辅助，以保证数据的一致性，因此共享简单。但是线程的数据同步要比进程略复杂。

# 选择

参考：

<https://www.cnblogs.com/zhanht/p/5401685.html>

具体是使用多进程还是多线程，还是单进程单线程，需要根据实际情况确认，业务场景是CPU密集型还是IO密集型，具体可以参考Redis的设计。

**需要频繁创建销毁的优先使用线程**，因为对进程来说创建和销毁一个进程代价是很大的。

线程的切换速度快，所以**在需要大量计算，切换频繁时用线程，还有耗时的操作使用线程可提高应用程序的响应**

因为对CPU系统的效率使用上线程更占优，所以可能要发展到多机分布的用进程，多核分布用线程;

**并行操作时使用线程**，如C/S 的服务器端并发线程响应用户的请求;

**需要更稳定安全时，适合选择进程；需要速度时，选择线程更好。**

# 总结

## 分析

从上面的分析可以看到，似乎线程有很多优势，比如，数据共享效率高，可应对并发操作，有效利用等待时间等等，但是多线程的编程比多进程要复杂，同时，多进程的可靠性较好，因为进程间不会相互影响。实际情况还是需要自己分析拿捏的。但是一般来说，实际应用中常常采用“进程+线程”结合的方式，而不是非此即彼，因为它们两者没有绝对的好与不好，而是适合于不同场景。

**线程上下文切换是有开销的，如果它的收益不能超过它的开销，那么使用多线程来提高效率将得不偿失。因此不要盲目推崇多线程**。如果为了提高效率采用多线程，那么线程中最多应为逻辑CPU数。也就是说如果你的程序绑在一个核上或者你只有一个CPU一个核，那么采用多线程只能提高同时处理的能力，而不能提高处理效率。

## 用户模式+内核模式

一般说来，一个进程在CPU上运行可以有两种运行模式，既可在用户模式下运行，又可在内核模式下运行（即进程分别工作在用户态和内核态，在内核态工作仍旧是这个进程，除非进行了进程的切换）。通常操作系统把虚拟地址空间划分为用户空间和内核空间，例如x86平台的Linux系统虚拟地址空间是0x00000000~0xffffffff，前3GB（0x00000000~0xbfffffff）是用户空间，后1GB（0xc0000000~0xffffffff）是内核空间。用户加载到用户空间，在用户模式下执行，不能访问内核中的数据，也不能跳转到内核代码中执行。这样可以保护内核，如果一个进程访问了非法地址，顶多这一个进程崩溃，而不会影响到内核和整个系统的稳定性。Cpu在产生中断或异常时不仅会跳转到中断或异常服务城西，还会自动切换模式，从用户模式切换到特权模式，因此从中断或异常程序可以跳转到内核代码中执行。事实上，整个内核就是由各种中断和异常处理程序组成的。即，正常情况下处理器在用户模式执行用户程序，在中断或异常情况下处理器切换到特权模式执行内核程序，处理完中断或异常之后再返回用户模式继续执行用户程序，例如，用户进程A调用了内核系统调用来获取当前的时钟滴答数，在执行用户进程A中的系统调用指令时会保存当前用户进程的IP，CS等当前状态，然后再跳转到内核空间（即内核代码区域）去执行像应的系统调用函数，获取当前的时钟滴答数。执行完后再通过IRET指令返回到进程A中（就是将进入时保存的信息再复位到相应的寄存器中），再接着从CS：EIP地址开始执行A进程的指令

进程在创建的时候除了创建进程的控制块之外，在内核里还创建了进程的内核栈，进程通过系统调用（例如fopen()或者open()）进入内核后，此时处理器处于特权级最高的（0级）内核代码中执行，当进程处于内核态时，执行的内核代码会使用当前进程的内核栈，是指向在进程的上下文上的，

内核模式的权限高于用户模式的权限。

用户级。系统用户可以与进行交互操作，如运行应用和系统命令，用户级通过系统调用接口访问内核级；内核级。操作系统自动运行一些功能，它们主要对硬件进行操作。