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。

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。

### 进程与程序



程序是存放在磁盘文件中的可执行文件。

进程时程序的执行实例。

### 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。

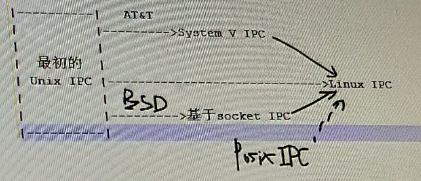
最常见的例子就是JDBC（它是同步阻塞的），这也是为什么很多人都说数据库是瓶颈的原因。这里的耗时其实是让CPU一直在等待I/O返回，说白了线程根本没有利用CPU去做运算，而是处于空转状态。而另外过多的线程，也会带来更多的ContextSwitch开销。

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。

## 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

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

#### 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

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。

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

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;

}

#### 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

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。

### 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

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。

## 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。

## 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

### 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。

### 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。

### 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？

### 如何设置线程数

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

# 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：**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。

# 选择

**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

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。

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。