1.综述

线程是进程的一个实体，是CPU调度和分派的基本单位，比进程更小的能独立运行的基本单位线程自己不拥有系统资源，只有一点不可缺少的资源，但是它可以与同属一个进程的其他线程共享进程所拥有的全部资源，具体包括：

一个栈

一个指向当前执行指令的指针

一个寄存器值的集合，定义了一部分描述正在执行线程的处理器状态的值

一个私有的数据区

1.1与进程的区别

将应用程序分解成可以准并行的多个顺序线程，并行实体共享同一个地址空间和所有可用数据的能力，而这正是多进程无法表达的；

线程更加轻量级，更快创建和销毁；

大量I/O和大量计算情况下，允许活动彼此重叠，更加快速；

进程用于资源集中在一起，线程则是在CPU上被调度的实体。

1.2优点

响应更快；

可以随时停止任务；

当前没有处理的任务，可已将处理器给其他任务；

占用大量处理事件的任务，可以定期的将处理器时间让给其他任务；

可以分别设置优先级。

1.3缺点

额外CPU开销；

等候一些独占性共享资源时会变慢；

死锁；

对共有变量的读和写，非原子性时会有不确定性

1.4适用

耗时或大量占有处理器的任务阻塞用户界面操作

各个任务必须等待外部资源

2.实现

2.1创建

int pthread\_create(pthread\_t \*restrict tidp,const pthread\_attr\_t \*restrict attr,void \*(\*start\_rtn)(void),void \*restrict arg)

pthread\_t id;

pthread\_create(&id,NULL,(void\*)myThread,NULL);

pthread\_t \*restrict tidp //要创建的线程的线程ID指针

const pthread\_attr\_t \*restrict attr //创建线程时的线程属性

void \*(\*start\_rtn)(void) //返回值是void类型的指针函数

void \*restrict arg start\_rtn 的形参

2.2终止

void pthread\_exit(void \*retval)

如果仅主线程调用pthread\_exit，只有主线程终止，进程及进程内其他线程继续存在，当所有线程终止时，进程也将终止

从执行函数返回，返回值是退出码

线程被同一进程的其他线程取消

特殊情况:当初始线程（即调用main的线程），从main调用返回或调用exit时，整个进程及其所有的线程将终止

2.3等待终止

int pthread\_join(thread\_t tid,void \*\*status)

如果多个线程等待一个线程完成，只有一个等待线程成功返回，其余的失败并返回ESRCH错误

仅适用于非分离的目标线程

2.4等待终止与终止

子线程使用exit 则可以结束整个进程；

使用join的线程会阻塞；

主线程结束，整个程序结束，所以主线程最好用join等子线程结束，使用一个join可以等多个线程；

主线程运行pthread\_exit结束主线程，不会结束子线程；

子线程最终要用pthread\_join或设置为分离状态来结束线程，否则资源不会被完全释放；

线程是分离的，则不需要pthread\_exit，线程结束自己释放所有资源；

线程是非分离的，线程会保留线程ID号，知道其他线程通过joining这个线程确认其已死亡。join的结果是joining线程得到已终止线程的退出状态，已终止下线程会消失；

线程自己运行结束，或调用pthread\_exit结束，会释放自己独有的空间资源。

3.多线程同步

3.1互斥锁

3.1.1初始化

int pthread\_mutex\_init(pthread\_mutex\_t\* mutex,const pthread\_mutexattr\_t\* mutexattr);

mutexattr:

PITHREAD\_MUTEX\_INITIALIZER

PITHREAD\_RECURSIVE\_MUTEX\_INITIALIZER\_NP

PTHREAD\_ERRORCHECK\_MUTEX\_INITIALIZER\_NP

3.1.2互斥操作函数

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

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

互斥被锁住时返回EBUSY，未上锁时上锁

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

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

3.1.3死锁

上锁的再次上锁

3.1.4设置互斥锁属性

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

三种：

快速型

同意线程给已上锁的上锁

错误检测型

已经上锁的，后续上锁将会失败，不会阻塞

kind:

PTHREAD\_MUTEX\_FAST\_NP

PTHREAD\_MUTEX\_RECURSIVE\_NP

PTHREAD\_MUTEX\_ERRORCHECK\_NP

3.2条件变量

int pthread\_cond\_init(pthread\_cond\_t \*cv,const pthread\_condattr\_t \*cattr);

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

int pthread\_cond\_timedwait(pthread\_cond\_t \*cv,pthread\_mutex\_t \*mp, const structtimespec \* abstime);

等待条件变量到指定时间

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

通知等待条件变量的单个线程

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

通知等待条件变量的所有线程

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

4.多线程属性

4.1属性结构

typedef struct

{

int detachstate; //线程的分离状态

int schedpolicy; //线程调度策略

struct sched\_param schedparam; //线程的调度参数

int inheritsched; //线程的继承性

int scope; //线程的作用域

size\_t guardsize; //线程栈末尾的警戒缓冲区大小

int stackaddr\_set;

void \* stackaddr; //线程栈的位置

size\_t stacksize; //线程栈的大小

}pthread\_attr\_t;

4.2属性初始化

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

int pthread\_attr\_destoroy(pthread\_attr\_t \*attr);

4.3线程分离

一个可结合的线程能够被其他线程收回其资源和杀死，在这之前，它的存储器资源是不释放;

用pthread\_join返回

一个分离的线程不能被其他线程回收或杀死，它的存储器资源在它终止时由系统自动释放;

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

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

detachstate:

PTHREAD\_CREATE\_DETACHED分离

PTHREAD\_CREATE\_JOINABLE正常

有些线程运行过快，可以用在线程中用pthread\_cond\_timewait来等待pthread\_create返回.

4.4线程继承

int pthread\_attr\_setinheritsched(pthread\_attr\_t \*attr,int inheritsched);

attr 线程属性变量

inheritsched 线程继承性

PTHREAD\_INHERIT\_SCHED表示新线程将继承创建线程的调度策略和参数

PTHREAD\_EXPLICIT\_SCHED 表示shiyongschedpolicy和schedparam属性中显示设置的调度策略和参数

int pthread\_attr\_getinheritsched(pthread\_attr\_t \*attr,int inheritsched);

4.5线程调度

int pthread\_attr\_setschedpolicy(const pthread\_attr\_t \*attr,int \*policy);

policy 调度策略

SCHED\_FIFO 先进先出

高优先级优先

SCHED\_RR 轮转法

如果一个SCHED\_RR策略的线程执行了超过一个固定时期没有阻塞，另外的SCHED\_RR或SCHED\_FIFO准备好时，将抢占运行的线程

SCHED\_OTHER 其他

int pthread\_attr\_getschedpolicy(const pthread\_attr\_t \*attr,int \*policy);

4.6调度参数

int pthread\_attr\_setschedparam(pthread\_attr\_t \*attr,const struct sched\_param \*param);

param 优先级

int sched\_get\_priority\_max (int policy);

int sched\_get\_priority\_min (int policy);

int pthread\_attr\_getschedparam(pthread\_attr\_t \*attr,const struct sched\_param \*param);

5.多线程小技巧

如何控制while循环的子线程的退出

设置while的判据变量为全局变量，如此可在子线程外，控制线程结束