十七、多线程

1. 基本概念

目前主流的操作系统支持多进程，而在每一个进程的内部，可以支持多线程，也就是说线程是进程中的程序流

进程是重量级单位，每个进程都需要独立的内存空间，启动新的进程对资源的是很大的消耗；而线程是轻量级单位，线程共享所在进程的内存空间，但是每个线程都有一块很小的独立栈区

比较进程和线程：

进程：

1. 进程是资源分配的基本单位。每个进程最少有一个主线程。
2. 进程都有自己的PCB
3. 父子进程只共享代码段

线程：

1. 线程是执行的基本单位。
2. 线程共享PCB(包括文件描述符等)和内存资源
3. 进程里的所有线程共享代码段、数据段和堆。每个线程都有自己的栈。
4. 线程里的数据段和堆都是临界资源，异步访问时会出错

临界资源：多个执行单位要同时访问的资源。

-------------------------------

2. 线程的创建

pthread\_create(3)

#include <pthread.h>

pthread\_t => unsigned long int(部分可能是结构体类型)

int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr,void \*(\*start\_routine) (void \*), void \*arg);

功能：创建一个新线程

第一个参数：值结果参数，存储新线程的tid

第二个参数：线程的属性，给NULL即可，表示默认属性

第三个参数：函数指针，表示新线程执行的处理函数

第四个参数：前面函数指针的参数

返回值：成功返回0，失败直接返回错误编号

---------------------

注意：

a.编译链接时，记得加上选项 -lpthread/-pthread //表示链接指定库文件

b.当启动新线程之后，子线程和主线程各自独立运行，每个线程内部的代码按照次序执行;也就是多线程之间相互独立，又相互影响,主线程结束,进程结束，而进程结束，进程中所有子线程随之结束

-----------------------

pthread\_self(3)

pthread\_t pthread\_self(void);

功能：获取正在运行线程的tid

----------------------------

3. 线程的终止和取消(自杀和他杀)

pthread\_exit(3)

void pthread\_exit(void \*retval);

功能：终止正在运行的线程

参数：值结果参数，用于存放退出状态信息，不能是栈里的数据(可以是代码段、数据段和堆)

与exit()/return比较：

---》pthread\_exit()：终止线程

---》exit()：终止进程

---》return：线程任务函数结束，线程还在

pthread\_cancel(3)

int pthread\_cancel(pthread\_t thread);

功能：对参数指定的线程发送取消的请求

此外，目标线程是否会被取消以及何时被取消，依赖于两个属性：state 和 type 的值

pthread\_setcancelstate(3)

int pthread\_setcancelstate(int state,int \*oldstate);

功能：设置新的取消状态，获取之前的取消状态

第一个参数：新的取消状态

PTHREAD\_CANCEL\_ENABLE - 允许被取消(默认的状态)

PTHREAD\_CANCEL\_DISABLE - 不允许被取消

第二个参数：获取之前的取消状态，不想获取给NULL即可

pthread\_setcanceltype(3)

int pthread\_setcanceltype(int type, int \*oldtype);

功能：设置新的取消类型，获取之前的取消类型

第一个参数：设置新的取消类型

PTHREAD\_CANCEL\_DEFERRED - 延迟取消到下一个取消点(默认的类型)

PTHREAD\_CANCEL\_ASYNCHRONOUS - 立即取消

第二个参数：获取旧的取消类型,不获取则给NULL即可

-------------------------

4. 线程的汇合(等待)和分离

pthread\_join(3) (相当于waitpid阻塞状态)

int pthread\_join(pthread\_t thread, void \*\*retval);

功能：等待一个线程的结束，并且获取退出码

第一个参数：指定线程的tid

第二个参数：值结果参数，拷贝存放退出码的内存地址(或任务函数返回值)，如果线程被取消则获得的数值为PTHREAD\_CANCELED

pthread\_detach(3)

int pthread\_detach(pthread\_t thread);

功能：将参数指定的线程标记为分离状态

分离状态：

---》线程终止后，其资源会立即被系统回收

---》不能被pthread\_join函数汇合，

---》即使别的线程使用pthread\_join函数等待，也不会阻塞(无效等待)

==============================

十八、线程同步

1. 相关问题

（1）同步与异步的区别：

同步：串行，不会有两个执行单位同时进行访问

异步：并行，会有多个执行单位同时访问同一个资源

（2）并发冲突与线程同步

---》并发冲突：当多个线程同时访问相同的共享资源时，如果不能相互协调配合，就难免会出现数据不一致的问题。

---》原因：代码中，单条指令在编译成汇编时，会生成多条指令(而只有这样的原子级指令才不会因为CPU的限时切换而被不断)，一组非原子化的操作常常会因为线程切换而导致未定义的结果。

---》线程同步：多线程同时运行时，有时需要互相排斥，有时需要相互协调。

----------------------------

2. 线程同步的解决方法之一：互斥锁

（1）互斥锁的使用步骤：

---》准备互斥锁(选择互斥锁的类型)

---》初始化互斥锁

---》使用共享资源前加锁

---》使用完后解锁

---》如果不再使用，则销毁互斥锁

（2）互斥锁相关函数

pthread\_mutex\_t mutex; //准备互斥锁(全局变量)

pthread\_mutex\_init(&mutex,NULL); //初始化mutex锁

pthread\_mutex\_lock(&mutex); //上锁，有锁等待

pthread\_mutex\_trylock(&mutex); //没锁上锁，有锁返回错误(非阻塞版本)

pthread\_mutex\_unlock(&mutex); //解锁

pthread\_mutex\_destroy(&mutex); //销毁互斥锁

补充：静态初始化互斥锁(等同于初始化时，第二个参数为NULL)

pthread\_mutex\_t fastmutex=PTHREAD\_MUTEX\_INITIALIZER;

--------------------------------

1. 线程的同步实现方法之二：条件变量(condition variable)

（1）作用：用于协调多线程之间的工作

（2）条件变量相关函数

pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER; //静态初始化一个条件变量

pthread\_cond\_init(3)

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

功能：初始化一个条件变量

参数： cond： 要初始化的条件变量；

cond\_attr： 指定为缺省属性，NULL

返回值：0成功；非0失败，代表的是错误码

pthread\_cond\_signal(3)

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

功能：向等待的一个线程发送重启信号(多个等待，不确定启动哪一个)

参数：cond：等待的条件变量

返回值：0成功；非0失败，代表的是错误码

pthread\_cond\_broadcast(3)

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

功能：给所有等待条件变量的线程发送信号(群发信号)

参数：cond：等待的条件变量

返回值：0成功；非0失败，代表的是错误码

pthread\_cond\_wait(3)

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

功能：等待某个条件变量

补充过程：a）（线程入口必须先加锁）自动解锁后，线程休眠 b）条件满足后重启线程并加锁

参数： cond： 等待的条件变量；

mutex： 需要用到的互斥锁

返回值：0成功；非0失败，代表的是错误码

pthread\_cond\_timedwait(3)

int pthread\_cond\_timedwait(pthread\_cond\_t \*cond,pthread\_mutex\_t \*mutex, const struct timespec \*abstime);

功能：等待某个条件变量，如果超时，返回错误，错误码ETIMEDOUT。

参数： cond: 等待的条件变量；

mutex: 需要用到的互斥锁

abstime: 等待的时间(结构体与gettimeofday(2)中参数类型相同)

返回值：0成功；非0失败，代表的是错误码

pthread\_cond\_destroy(3)

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

功能：销毁条件变量

参数：cond:要销毁的条件变量

返回值：0成功；非0失败，代表的是错误码

（3）举例：生产者和消费者模型（重点掌握）

生产者生产一个结构体，将结构体插入到链表的头部。

消费者从链表的头部取出结构体，进行消费。

代码参见：pthread\_cond.c

----------------------------

4. 线程的同步实现方法之三：信号量(Semaphore)

（1）基本概念

信号量 - 本质就是一个由系统内核维护的全局计数器，用于控制同时访问共享资源的进程数/线程数

---》当信号量最大值是1时，效果等同于互斥量

（2）信号量的使用流程

---》定义全局信号量 - sem\_t sem;

---》初始化信号量 - sem\_init(&sem, 0, 预设资源值);

---》获取一个信号量(减1) - sem\_wait(&sem);

---》使用共享资源

---》释放一个信号量(加1) - sem\_post(&sem);

---》如果不再使用，则销毁信号量 - sem\_destroy(&sem);

（3）信号量集：信号量的集合，就是计数器的集合。

----------------------------

（4）操作信号量的相关函数

#include <semaphore.h>

sem\_t sem; //定义信号量

sem\_init(3)

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

功能：初始化信号量

参数：

sem：要初始化的信号量

pshared:

0 - 信号量是线程间的共享资源

非0 - 信号量，子进程可以访问父进程的内存映射(或者定义在其他进程的共享内存空间)

value：预先设置的信号量中的资源数量

返回值：0成功；-1失败，errno被设置

sem\_wait(3)(阻塞)/sem\_trywait(3)(非阻塞)

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

功能：使信号量的值-1，如果信号量的值为0，阻塞。(实质：等待直至占用一个空间)

参数：sem: 指定要操作的信号量

返回值：0成功；-1失败，errno被设置

sem\_post(3)

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

功能：释放资源，给指定的信号量加1。(实质：解除一个占用一个空间)

参数：sem：指定的信号量

返回值：0成功；-1失败，errno被设置

sem\_destroy(3)

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

功能：销毁信号量

参数：sem：指定销毁的信号量

返回值：0成功；-1失败，errno被设置

举例：使用环形队列实现生产者和消费者模型

代码参见：pthread\_sem.c