# 线程

再论进程

多进程同时实现读取键盘和鼠标

使用多进程直接实现了并发，并没有使用select函数。

char buf[200];

int main(void){

//思路就是创建一个子进程，然后父子进程分别进行读键盘和鼠标的工作

int ret =-1;

int fd=-1;

ret=fork();

if(ret==0){

//子进程

fd=open("/dev/input/mouse1",O\_RDONLY);//这里还是要设置成阻塞方式，两个进程， 不要占用cpu。

if(fd<0){

perror("open error");

return -1;

}

while(1){

memset(buf,0,sizeof(buf));

read(fd,buf,50);

printf("从鼠标读了[%s].\n",buf);

}

}else if(ret>0){

//父进程

while(1){

memset(buf,0,sizeof(buf));

read(0,buf,50);

printf("从键盘读出了[%s].\n",buf);

}

}else{

perror("fork:");

}

}

多线程 使用线程事先鼠标和键盘的同时读取

#include<stdio.h>

#include<pthread.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include<unistd.h>

#include<string.h>

char buf[200];

void \* func(void \* arg){

while(1){

memset(buf,0,sizeof(buf));

read(0,buf,50);

printf("从键盘读出[%s].\n",buf);

}

}

int main(){

int ret=-1;

int fd=-1;

pthread\_t th=-1;

ret=pthread\_create(&th,NULL,func,NULL);

if(ret!=0){

printf("pthread\_create error.\n");

return -1;

}

//因为主线程是while(1)死循环，所以在这里可以使用pthread\_detach分离子线程

//主任务

fd=open("/dev/input/mouse1",O\_RDONLY);

if(fd<0){

perror("open error.\n");

return -1;

}

while(1){

memest(buf,0,sizeof(buf));

read(fd,buf,50);

printf("从鼠标读出了[%s].\n",buf);

}

}

gcc pthread.c -o pthread.o -l pthread

./pthread.o

在多核心CPU（对称多处理器架构SMP）架构下效率最大化

线程常见函数

线程的创建与回收

pthread\_create //主线程用来创造子线程

pthread\_join //主线程用来等待（阻塞）回收子线程

pthread\_detach //主线程用来分离子线程，分离后主线程不再回收子线程

上两个函数二选一

线程取消

pthread\_cancel 一个线程取消另一个子线程；一般都是主线程调用该方法取消子线程

pthread\_setcancelstate 子线程设置自己是否允许被取消

pthread\_setcanceltype 在上个函数设置成enable（允许取消）这个函数才有意义。立刻被取消还是等待某些程序执行完成后才被取消。

线程函数退出机制

pthread\_exit与return退出

pthread\_cleanup\_push

pthread\_cleanup\_pop

if(cnt==0){

cnt++;

pthread\_cleanup\_push(function,arg);

//子线程操作

//子线程有可能在这里被主线程取消

pthread\_cleanup\_pop(1)；

cnt--;

}

function(){

cnt--;

}

获取线程id

pthread\_self

线程同步之信号量

任务：用户从终端输入任意字符然后然后统计个数显示，输入end结束

使用多线程实现：主线程获取用户输入并判断是否退出，子线程计数

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#include<pthread.h>

#include<semaphore.h>

char buf[200]={0};

sem\_t sem;

//子线程程序，统计buf的字符个数并打印

void func(void \*arg){

//子线程首先应该有个循环

//循环中阻塞在等待主线程激活的时候，子线程被激活后就去获取buf中的字符

//长度，然后打印，完成后再次被阻塞。

sem\_wait(&sem);

while(strncmp(buf,"end",3)!=0){

printf("本次输入了%d个字符.\n",strlen(buf));

memset(buf,0,sizeof(buf));

sem\_wait(&sem);

}

pthread\_exit(NULL);

}

int main(void){

int ret=-1;

pthread\_t th=-1;

sem\_init(&sem,0,0);

ret=pthread\_create(&th,NULL,func,NULL);

if(ret!=0){

printf("线程创建失败.\n");

exit(0);

}

printf("输入一个字符串以回车结束\n");

while(scanf("%s",buf)){

if(strncmp(buf,"end",3)==0){

printf("程序结束.\n");

//exit(0);

sem\_post(&sem);

break;

}

sem\_post(&sem);

//主线程在收到用户输入的字符串，并确认不是end后

//就发信号激活子线程计数

//子线程被阻塞，主线程可以激活，这就是线程的同步问题

//printf("本次输入了%d个字符\n",strlen(buf));

}

printf("等待回收子线程.\n");

pthread\_join(th,NULL);

if(ret!=0){

printf("pthread\_join error.\n");

exit(-1);

}

printf("子线程回收成功.\n");

sem\_destroy(&sem);

return 0;

}

线程同步之互斥锁

互斥锁又叫互斥量（mutex）

相关函数：pthread\_mutex\_init pthread\_mutex\_detroy

pthread\_mutex\_lock pthread\_mutex\_unlock

互斥锁和信号量的关系：可以认为互斥锁是一种特殊的信号量

互斥锁主要用于实现关键段保护

用互斥锁实现上节的代码

注意：pthread\_mutex\_init时提示找不到函数，说明你没有安装pthread相关的man手册，安装方法：sudo apt-get install manpages-posix-dev

上锁

访问

解锁

以下程序视频中仍然有逻辑问题，还是会存在输入一次打印两次的问题，主要在sleep上有逻辑问题。

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#include<pthread.h>

#include<semaphore.h>

char buf[200]={0};

pthread\_mutex\_t mutex;

//子线程程序，统计buf的字符个数并打印

void func(void \*arg){

//子线程首先应该有个循环

//循环中阻塞在等待主线程激活的时候，子线程被激活后就去获取buf中的字符

//长度，然后打印，完成后再次被阻塞。

sleep(1);

while(strncmp(buf,"end",3)!=0){

pthread\_mutex\_lock(&mutex);

printf("本次输入了%d个字符.\n",strlen(buf));

memset(buf,0,sizeof(buf));

pthread\_mutex\_lock(&mutex);

sleep(1);

}

pthread\_exit(NULL);

}

int main(void){

int ret=-1;

pthread\_t th=-1;

pthread\_mutex\_init(&mutex,NULL);

ret=pthread\_create(&th,NULL,func,NULL);

if(ret!=0){

printf("线程创建失败.\n");

exit(0);

}

printf("输入一个字符串以回车结束\n");

while(1){

pthread\_mutex\_lock(&mutex);

scanf("%s",buf)

pthread\_mutex\_unlock(&mutex);

if(!strncmp(buf,"end",3)){

printf("程序结束.\n");

//exit(0);

break;

}

sleep(1)

//主线程在收到用户输入的字符串，并确认不是end后

//就发信号激活子线程计数

//子线程被阻塞，主线程可以激活，这就是线程的同步问题

//printf("本次输入了%d个字符\n",strlen(buf));

}

printf("等待回收子线程.\n");

pthread\_join(th,NULL);

if(ret!=0){

printf("pthread\_join error.\n");

exit(-1);

}

printf("子线程回收成功.\n");

pthread\_mutex\_destroy(&mutex);

return 0;

}

gcc mutex.c -o mutex.o -pthread //建议使用-pthread 而不是-lpthread

线程同步之条件变量

相关函数：pthread\_cond\_init pthread\_cond\_destroy

phread\_cond\_wait //条件成立则唤醒

pthread\_cond\_signal/pthread\_cond\_broadcast

使用条件变量实现上节代码

使用条件变量之前必须要先上锁

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#include<pthread.h>

#include<semaphore.h>

char buf[200]={0};

pthread\_mutex\_t mutex;

pthread\_cond\_t cond;

//子线程程序，统计buf的字符个数并打印

void func(void \*arg){

//子线程首先应该有个循环

//循环中阻塞在等待主线程激活的时候，子线程被激活后就去获取buf中的字符

//长度，然后打印，完成后再次被阻塞。

while(strncmp(buf,"end",3)!=0){

pthread\_mutex\_lock(&mutex);

pthread\_cond\_wait(&cond,&mutex);

printf("本次输入了%d个字符.\n",strlen(buf));

memset(buf,0,sizeof(buf));

pthread\_mutex\_lock(&mutex);

}

pthread\_exit(NULL);

}

int main(void){

int ret=-1;

pthread\_t th=-1;

pthread\_mutex\_init(&mutex,NULL);

pthread\_cond\_init(&cond,NULL);

ret=pthread\_create(&th,NULL,func,NULL);

if(ret!=0){

printf("线程创建失败.\n");

exit(0);

}

printf("输入一个字符串以回车结束\n");

while(1){

pthread\_cond\_signal(&cond);

scanf("%s",buf)

if(!strncmp(buf,"end",3)){

printf("程序结束.\n");

//exit(0);

break;

}

//主线程在收到用户输入的字符串，并确认不是end后

//就发信号激活子线程计数

//子线程被阻塞，主线程可以激活，这就是线程的同步问题

//printf("本次输入了%d个字符\n",strlen(buf));

}

printf("等待回收子线程.\n");

pthread\_join(th,NULL);

if(ret!=0){

printf("pthread\_join error.\n");

exit(-1);

}

printf("子线程回收成功.\n");

pthread\_mutex\_destroy(&mutex);

pthread\_cond\_destroy(&cond);

return 0;

}