

系列课程 —Linux系统编程

第六章

Linux 线程

讲师：任继梅

QQ：59189174

课前提问

- 1.什么是线程?
- 2.如何创建线程?
- 3.线程和进程有何区别?
- 4.如何实现把自己的程序做成多个任务一起
并发协作运行完成工作?

本章内容

✓ 6.1 线程的概念



✓ 6.2 多线程编程



✓ 6.3 线程的同步



✓ 6.4 线程的互斥



✓ 6.5 操作系统和实时系统



✓ 6.6 线程和进程



本章目标

- ✓ 线程的概念 ★★★★★
- ✓ 多线程编程 ★★★★★
- ✓ 线程的同步 ★★★★★
- ✓ 线程的互斥 ★★★★★
- ✓ 操作系统和实时系统 ★★★★★
- ✓ 线程和进程 ★★★★★

第一节

线程概念

线程概念

线程概念

- 每个用户进程有自己的地址空间
- 系统为每个用户进程创建一个task_struct来描述该进程
- 该结构体中包含了一个指针指向该进程的虚拟地址空间映射表
- 实际上task_struct 和地址空间映射表一起用来表示一个进程



task_struct

The diagram consists of a green oval shape. Inside the oval, there is an orange rectangular box labeled 'task_struct' and a purple rectangular box labeled 'Memory_map'.

Memory_map

线程概念

线程概念

- 由于进程的地址空间是私有的，因此在进程间上下文切换时（进程调度），系统开销比较大
- 创建新的进程时，基本copy父进程，内存开销也比较大
- 为了提高系统的性能，许多操作系统规范里引入了轻量级进程的概念，也被称为线程，Linux的内核结构中只存在轻量级进程，进程和线程的区别只是在clone 时是否设置为共享内存，二者在内核中存储的数据结构是一样的，而系统调度的单位也是轻量级进程。
- 在同一个进程中创建的线程共享该进程的地址空间
- Linux里同样用task_struct来描述一个线程。线程和进程都参与统一的调度

线程概念

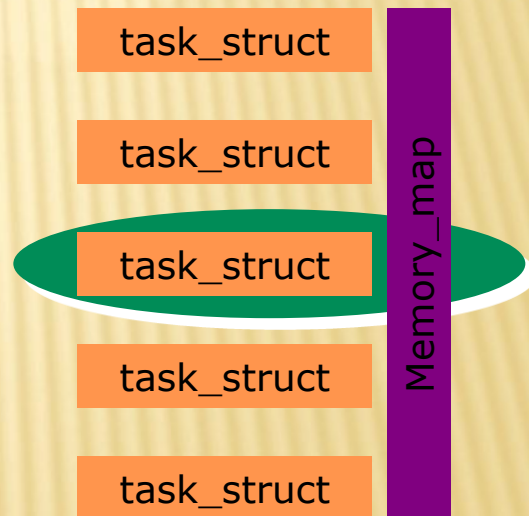
线程概念

通常线程指的是共享相同地址空间的多个任务

使用多线程的好处：

大大提高了任务切换的效率

优化了内存使用的销量



线程概念

线程概念

- ❑ 多线程通过第三方的线程库来实现
- ❑ Native POSIX Thread Library (NPTL)
- ❑ 是早期Linux Threads的改进
- ❑ 显著的提高了运行效率
 - ❑ 尽管是基于进程的实现，但新版的NPTL创建线程的效率非常高。一些测试显示，基于NPTL的内核创建10万个线程只需要2秒，而没有NPTL支持的内核则需要长达15分钟。
- ❑ NPTL是一个1×1的线程模型，即一个线程对应于一个操作系统的调度进程，优点是非常简单。

线程概念

一个进程中的多个线程共享以下资源

- ❑ 可执行的指令
- ❑ 静态数据
- ❑ 进程中打开的文件描述符
- ❑ 信号处理函数
- ❑ 当前工作目录
- ❑ 用户ID
- ❑ 用户组ID

线程概念

每个线程私有的资源如下

- ❏ 线程ID (TID)
- ❏ PC(程序计数器)和相关寄存器
- ❏ 堆栈
- ❏ 局部变量
- ❏ 返回地址
- ❏ 错误号 (errno)
- ❏ 信号掩码和优先级
- ❏ 执行状态和属性

线程概念

线程ID

- ❑ 线程有ID, 但不是系统唯一, 而是进程环境中唯一有效.
- ❑ 线程的句柄是pthread_t类型, 该类型不能作为整数处理, 而是一个结构.

```
#include <pthread.h>
```

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

- ❑ 返回值: 相等返回非0, 不相等返回0.
- ❑ 说明: 比较两个线程ID是否相等.

```
#include <pthread.h>
```

```
pthread_t pthread_self(void);
```

- ❑ 返回值: 返回调用线程的线程ID.

线程概念

例1:

```
#include <stdio.h>
#include <pthread.h>

int main()
{
    pthread_t tid;

    /*pthread_self()获取当前线程的id*/
    tid = pthread_self();

    printf("tid = %lu\n", (unsigned long)tid);

    return 0;
}

/*gcc 01pthread.c -lpthread
-lpthread指定使用pthread线程库
*/
```

◆ 编译多线程程序

```
# gcc -o sample sample.c -lpthread -  
D_REENTRANT
```

-lpthread : 链接pthread库




-D_REENTRANT : 生成可重入代码

第二节




多线程编程

多线程编程

NPTL线程库中提供了如下基本操作

-  创建线程
-  删除线程
-  控制线程

线程间同步和互斥机制

-  信号量
-  互斥锁
-  条件变量

多线程编程

创建新的线程

```
#include <pthread.h>
```

```
int pthread_create(pthread_t *restrict thread,  
                  const pthread_attr_t *restrict attr,  
                  void *(*start_routine)(void*),  
                  void *restrict arg);
```

- @thread 为新线程的id
- @attr 为线程属性
- @start_routine为新线程的任务
- @arg 为传递为新线程的数据
- 返回值:
 - 成功返回0
 - 失败返回-1

多线程编程

等待指定的线程退出

```
#include <pthread.h>
```

```
int pthread_join(pthread_t thread, void **value_ptr);
```

- @thread 为指定线程id
- @value_ptr 为指定线程的返回状态
- 返回值:
 - 成功返回0
 - 失败返回-1

多线程编程

返回例子

```
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void *my_handler(void *data)
{
    int *b = (int *)data;

    printf("hello new pthread...b = %d\n", *b);

    int *a = (int *)malloc(sizeof(int));

    *a = 321;

    //返回地址
    return a;
}
```

多线程编程

返回例子

```
int main()
{
    int ret;
    pthread_t tid;
    void *p;

    int a = 1000;
    ret = pthread_create(&tid, NULL, my_handler, (void *)&a);

    pthread_join(tid, &p); //接收地址

    printf("a = %d\n", a);

    printf("return value = %d\n", *(int *)p);

    free(p);

    return 0;
}
```


多线程编程

线程退出

```
#include <pthread.h>
```

```
void pthread_exit(void *value_ptr);
```

- @value_ptr 为线程的返回状态值

取消线程

```
#include <pthread.h>
```

```
int pthread_cancel(pthread_t thread);
```

- 返回值:
 - 成功返回0
 - 失败返回-1

多线程编程-示例1:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

char message[32] = "Hello World";
void *thread_function(void *arg);

int main(int argc, char *argv[])
{
    int ret;
    pthread_t a_thread;
    void *thread_result;

    /*使用缺省属性创建线程*/
    ret = pthread_create(&a_thread, NULL, thread_function,
                        (void *)message);
    if (ret < 0)
    {
        perror("fail to pthread_create");
        return -1;
    }
}
```

多线程编程-示例1:

```
printf("waiting for thread to finish\n");
/*等待线程结束*/
ret = pthread_join(a_thread, &thread_result);
if (ret < 0)
{
    perror("fail to pthread_join");
    return -1;
}
printf("MESSAGE is now %s\n", message);

return 0;
}

void *thread_function(void *arg)
{
    printf("thread_function is running, arg is %s\n", (char *)arg);
    strcpy(message, "marked by thread");

    pthread_exit("Thank you for the cpu time");
}
```


多线程编程-示例2:

//1. 创建一个新的线程, 传递给新线程一个数字()
//2. 在新线程中 判断 这个数字 是否为 质数
//3. 返回是否是质数()

```
#include <stdio.h>
#include <pthread.h>
```

```
struct data
{
    int start;
    int n;
};
```

```
void *func(void *data)
{
    struct data *p = (struct data *)data;
    int i = 2;
    int flag = 0;
    int num;
```

多线程编程-示例2:

```
for(num = p->start; num < (p->start+p->n); num++)
{
    flag = 0;
    for(i = 2; i < num; i++)
    {
        if(num % i == 0)
        {
            flag = 1;
            break;
        }
    }

    if(flag == 1)
    {
        //printf("%d is not prime\n", num);
    }else
    {
        printf("%d is prime\n", num);
    }
}

return NULL;
}
```

多线程编程-示例2:

```
int main()
{
    int ret;
    pthread_t tid;
    void *retval;

    struct data d = {200000000, 100};

    ret = pthread_create(&tid, NULL, func, (void *)&d);
    if(ret != 0)
    {
        printf("pthread_create error\n");
        return -1;
    }

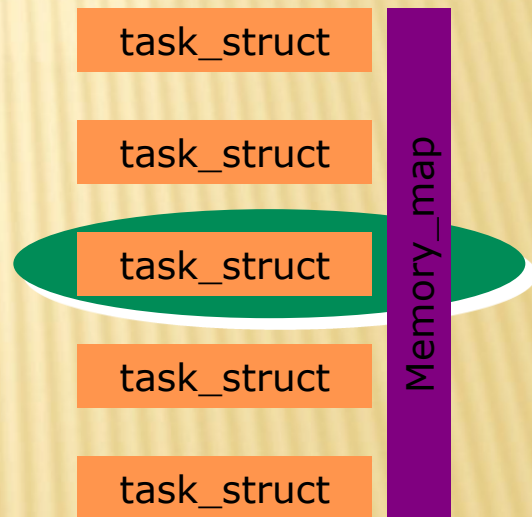
    pthread_join(tid, &retval);

    return 0;
}
```


多线程编程

多线程共享同一个进程的地址空间

- 优点：线程间很容易进行通信
 - 通过全局变量实现数据共享和交换
- 缺点：多个线程同时访问共享对象
 - 时需要引入同步和互斥机制



第三节

线程同步

线程互斥

- ❖ 引入互斥(mutual exclusion)锁的目的是用来保证共享数据操作的完整性。
- ❖ 互斥锁主要用来保护临界资源
- ❖ 每个临界资源都由一个互斥锁来保护，任何时刻最多只能有一个线程能访问该资源
- ❖ 线程必须先获得互斥锁才能访问临界资源，访问完资源后释放该锁。如果无法获得锁，线程会阻塞直到获得锁为止

POSIX MUTEX API

初始化互斥锁

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t *restrict attr);

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- mutex: 互斥锁
- attr: 互斥锁属性 // NULL表示缺省属性
- 返回值: 成功返回0, 失败返回-1

Posix Mutex API

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 pthread_mutex_unlock(pthread_mutex_t *mutex);
```

❏ mutex: 互斥锁

❏ 返回值: 成功返回0, 失败返回-1

线程互斥示例

```
#include <stdio.h>
#include <pthread.h>

//1.定义并初始化线程锁
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

int n = 200000000;

void *func(void *data)
{
    int i = 2;
    int flag = 0;
    int num = 0;

    while(1)
    {
        flag = 0;

        //如果没有锁上, 就会加锁
        //如果已经锁上, 就会等待, 直到解锁
        pthread_mutex_lock(&lock);
        //取n
        num = n;
```


线程互斥示例

```
if(n == 200000401)
{
    pthread_mutex_unlock(&lock);
    break;
}
//ldr r0, [&n]
//add r0,r0,#1
//str r0, [&n]
n = n + 1;
pthread_mutex_unlock(&lock);

for(i = 2; i< num; i++)
{
    if(num % i == 0)
    {
        flag = 1;
        break;
    }
}

if(flag == 1)
{
    //printf("%d is not prime\n", num);
}
```

线程互斥示例

```
    else
    {
        printf("%d is prime\n", num);
    }

}

return NULL;
}
```

线程互斥示例

```
int main()
{
    int ret;
    int i;
    pthread_t tid[5];
    void *retval;

    for(i = 0; i < 5; i++)
    {
        ret = pthread_create(&tid[i], NULL, func, NULL);
        if(ret != 0)
        {
            printf("pthread_create error\n");
            return -1;
        }
    }

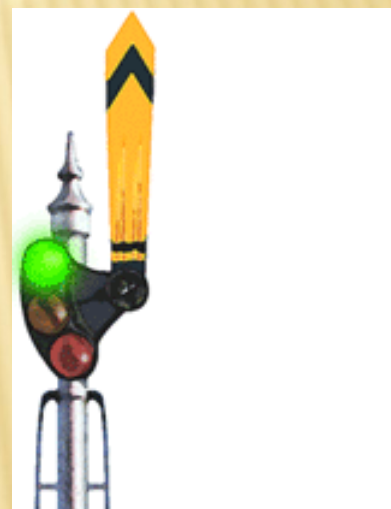
    for(i = 0; i < 5; i++)
    {
        pthread_join(tid[i], &retval);
    }

    return 0;
}
```


线程同步

线程同步

- 同步(synchronization)指的是多个任务(线程)
- 按照约定的顺序相互配合完成一件事情
- 1968年, Edsgar Dijkstra 基于信号量的概念提出了一种同步机制
- 由信号量来决定线程是继续运行还是阻塞等待



线程同步-P/V操作

P/V操作

- ❑ 信号量代表某一类资源，其值表示系统中该资源的数量
- ❑ 信号量是一个受保护的变量，只能通过三种操作来访问
 - ❑ 初始化
 - ❑ P 操作(申请资源)
 - ❑ V 操作(释放资源)
- ❑ 信号量的值为非负整数

线程同步-P/V操作

P/V操作

❏ P(S) 含义如下:

if (信号量的值大于0)

{

 申请资源的任务继续运行;

 信号量的值减一; }

else { 申请资源的任务阻塞; }

❏ V(S) 含义如下:

if (没有任务在等待该资源){ 信号量的值加一; }

else { 唤醒第一个等待的任务, 让其继续运行 }

POSIX SEMAPHORE API

- ✗ posix中定义了两类信号量：
 - + 无名信号量(基于内存的信号量)
 - + 有名信号量
- ✗ pthread库常用的信号量操作函数如下：
 - + `int sem_init(sem_t *sem, int pshared, unsigned int value)`
 - + `int sem_wait(sem_t *sem);` // P操作
 - + `int sem_post(sem_t *sem);` // V操作
 - + `int sem_trywait(sem_t *sem);`
 - + `int sem_getvalue(sem_t *sem, int *svalue);`



Posix Semaphore API

人 初始化信号量

```
#include <semaphore.h>
```

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

Link with `-lrt` or `-pthread`

- sem: 初始化的信号量
- pshared: 信号量共享的范围(0: 线程间使用 非0:进程间使用)
- value: 信号量初值
- 成功返回0, 失败返回-1

Posix Semaphore API

V操作

```
#include <semaphore.h>
```

```
int sem_post(sem_t *sem);
```

 Link with `-lrt` or `-pthread`.

- 信号量

- 成功返回0，失败返回-1

Posix Semaphore API

人 P操作

```
#include <semaphore.h>
```

```
int sem_wait(sem_t *sem);
```

```
int sem_trywait(sem_t *sem);
```

```
int sem_timedwait(sem_t *sem,  
                  const struct timespec *abs_timeout);
```

● Link with `-lrt` or `-pthread`.

⊞ 信号量

⊞ 成功返回0，失败返回-1

线程同步-示例

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>
```

```
char buf[60];
void *function(void *arg);
sem_t sem;
```

```
void *function(void *arg)
{
    while(1)
    {
        sem_wait(&sem);
        printf("You enter %lu characters\n", strlen(buf) - 1);
    }
}
```

线程同步-示例

```
int main(int argc, char *argv[])
{
    int ret;
    pthread_t tid;

    if(pthread_create(&tid, NULL, function, NULL) < 0)
    {
        perror("fail to pthread_create\n");
        exit(-1);
    }
    ret = sem_init(&sem, 0, 0);
    if(ret < 0)
    {
        perror("fail to sem_init\n");
        exit(-1);
    }
    do {
        printf(">");
        fflush(stdout);
        fgets(buf, 60, stdin);
        sem_post(&sem);
    } while (strncmp(buf, "quit", 4) != 0);
    return 0;
}
```


第四节

线程总结

进程和线程

● 接口比较

进程接口	线程接口	描述
fork	pthread_create	创建新的任务程序流
exit	pthread_exit	从任务程序流中退出
waitpid	pthread_join	对程序流进行善后，获取退出码
atexit	pthread_cleanup_push	注册程序流退出时的清理函数
getpid	pthread_self	获取程序流的ID
abort	pthread_cancel	非正常退出程序流

进程和线程

● 特点比较

进程	线程
可以通过exec执行不同的程序	多个线程执行程序的一部分
拥有独立的地址空间和资源	所有线程共享地址空间和资源
粗略的并行任务	精细的并行任务
使用进程间通信的方式	共享数据很容易

进程和线程

● 使用线程的经验

- ◆ 处理耗时操作
- ◆ 必须并行处理
- ◆ 多核处理器时加速
- ◆ linux线程并不完全遵循POSIX标准
- ◆ 线程里不要处理信号
- ◆ 没有必要使用时不要为了显摆故显高深刻意为之
- ◆ 不要在多线程程序里单独调用exec，会导致所有线程消失
- ◆ 可以调用fork，但是此时子进程仅限于本线程，子进程只存在一个线程，但锁的处理很麻烦，除非立即调用exec
- ◆ 线程中不是所有的系统调用函数都能安全运行的（线程安全函数）
- ◆ 时刻关注共享

课程总结

● 本节课程内容

- 线程的概念
- 线程的创建
- 多线程编程
- 线程同步
- 线程互斥
- 操作系统和实时系统
- 线程和进程

● 下节课

- 进程通讯
- 消息传递
- 共享内存

联系方式

QQ: 59189174

E-mail: yumeifly@sohu.com