第八章：线程

8.1 线程概述

8.1.1 线程的概念

每个进程都拥有自己的数据段、代码段和堆栈段，这就造成进程在进行创建、切换、撤销操作时，需要较大的系统开销。

为了减少系统开销，从进程中演化出了线程。

线程存在于进程中，共享进程的资源。

线程是进程中的独立控制流，由环境（包括寄存器组和程序计数器）和一系列的执行指令组成。

每个进程有一个地址空间和一个控制线程。

8.1.2 线程和进程的比较

调度：

线程是 CPU 调度和分派的基本单位。

拥有资源：

进程是系统中程序执行和资源分配的基本单位。

线程自己一般不拥有资源（除了必不可少的程序计数器，一组寄存器和栈），但它可以去访问其所属进程的资源，如进程代码段，数据段以及系统资源（已打开的文件，I/O 设备等）。

系统开销：

同一个进程中的多个线程可共享同一地址空间，因此它们之间的同步和通信的实现也变得比较容易。

在进程切换时候，涉及到整个当前进程 CPU 环境的保存以及新被调度运行的进程的 CPU 环境的设置；而线程切换只需要保存和设置少量寄存器的内容，并不涉及存储器管理方面的操作，从而能更有效地使用系统资源和提高系统的吞吐量。

并发性：

不仅进程间可以并发执行，而且在一个进程中的多个线程之间也可以并发执行。

8.1.3 多线程的用处

使用多线程的目的主要有以下几点：

多任务程序的设计

一个程序可能要处理不同应用，要处理多种任务，如果开发不同的进程来处理，系统开销很大，数据共享，程序结构都不方便，这时可使用多线程编程方法。

并发程序设计

一个任务可能分成不同的步骤去完成，这些不同的步骤之间可能是松散耦合，可能通过线程的互斥，同步并发完成。这样可以为不同的任务步骤建立线程。

网络程序设计

为提高网络的利用效率，我们可能使用多线程，对每个连接用一个线程去处理。

数据共享

同一个进程中的不同线程共享进程的数据空间，方便不同线程间的数据共享。

在多 CPU 系统中，实现真正的并行。

8.2 线程的基本操作

就像每个进程都有一个进程号一样，每个线程也有一个线程号。
进程号在整个系统中是唯一的，但线程号不同，线程号只在它所属的进程环境中有效。
进程号用 pid_t 数据类型表示，是一个非负整数。线程号则用 pthread_t 数据类型来表示。
有的系统在实现 pthread_t 的时候，用一个结构体来表示，所以在可移植的操作系统实现不能把它做为整数处理。

8.2.1 线程的创建

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

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

功能：

创建一个线程。

参数：

thread：线程标识符地址。

attr：线程属性结构体地址。

start_routine：线程函数的入口地址。

arg：传给线程函数的参数。

返回值：

成功：返回 0

失败：返回非 0

与 fork 不同的是 pthread_create 创建的线程不与父线程在同一点开始运行，而是从指定的函数开始运行，该函数运行完后，该线程也就退出了。

线程依赖进程存在的，如果创建线程的进程结束了，线程也就结束了。

线程函数的程序在 pthread 库中，故链接时要加上参数-lpthread。

例：01_pthread_create_1.c 验证多线程实现多任务，及线程间共享全局变量

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

int var   = 8;

void *thread_1(void *arg)
{
    while(1)
    {
        printf("this is my new thread1 var++\n");
        var++;
        sleep(1);
    }
    return NULL;
}

void *thread_2(void * arg)
{
    while(1)
    {
        printf("this is my new thread2 var = %d\n", var);
```

```
        sleep(1);
    }
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t tid1,tid2;

    /*创建两个线程*/
    pthread_create(&tid1, NULL, thread_1, NULL);
    pthread_create(&tid2, NULL, thread_2, NULL);
    while(1);
    return 0;
}
```

例 01_pthread_create_2.c 验证线程函数传参

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

/*传参方法 1*/
void *thread_1(void *arg)
{
    int rec = 0;

    sleep(1);
    rec = (int)(arg);
    printf("new thread1 arg = %d\n", rec);
    return NULL;
}

/*传参方法 2*/
void *thread_2(void * arg)
{
    int rec = 0;

    sleep(1);
```

```
    rec = *((int *)arg);
    printf("new thread2 arg = %d\n", rec);
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t tid1,tid2;
    int test = 100;
    //double test = 100;

    /*创建两个线程*/
    pthread_create(&tid1, NULL, thread_1, (void *)test);
    pthread_create(&tid2, NULL, thread_2, (void *)(&test));
    //test++;
    while(1);
    return 0;
}
```

8.2.2 线程等待

#include <pthread.h>
int pthread_join(pthread_t thread, void **retval);

功能：

等待子线程结束，并回收子线程资源。

参数：

thread：被等待的线程号。

retval：用来存储线程退出状态的指针的地址。

返回值：

成功返回 0，失败返回非 0。

例：02_pthread_join_1.c　验证 pthread_join 的阻塞效果

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

void *thead(void *arg)
{
    static int num = 123;
```

```
    printf("after 2 seceonds, thr      ead will return\n");
    sleep(2);
    return &num;
}

int main(int argc, char *argv[])
{
    pthread_t tid1;
    int ret = 0;
    void *value = NULL;

    ret = pthread_create(&tid1, NULL, thead, NULL);
    if(ret != 0)
        perror("pthread_create");
    pthread_join(tid1, &value);
    printf("value = %d\n", *((int *)value));
    return 0;
}
```

例：02_pthread_join_2.c　验证 pthread_join 接收线程的返回值

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

void *thead(void *arg)
{
    int num = 123;

    printf("after 2 seceonds, thread will return\n");
    sleep(2);
    return (void *)num;
}

int main(int argc, char *argv[])
{
    pthread_t tid1;
    int ret = 0;
    int value = 0;
```

```
    ret = pthread_create(&tid1, NULL, thead, NULL);
    if(ret != 0)
        perror("pthread_create");
    pthread_join(tid1, (void *)(&value));
    printf("value = %d\n", value);
    return 0;
}
```

8.2.3 线程分离

线程的结合态和分离态

linux 线程执行和 windows 不同，pthread 有两种状态：
可结合的（joinable）或者是分离的（detached），线程默认创建为可结合态。
如果线程是 joinable 状态，当线程函数自己返回退出时或 pthread_exit 时都不会释放线程所占用堆栈和线程描述符（总计 8K 多）。只有当你调用了 pthread_join 之后这些资源才会被释放。
若是 detached 状态的线程，这些资源在线程函数退出时或 pthread_exit 时自动会被释放，使用 pthread_detach 函数将线程设置为分离态。

创建一个线程后应回收其资源，但使用 pthread_join 函数会使调用者阻塞，故 Linux 提供了线程分离函数：pthread_detach。

#include <pthread.h>
int pthread_detach(pthread_t thread);

功能：

使调用线程与当前进程分离，使其成为一个独立的线程，该线程终止时，系统将自动回收它的资源。

参数：

thread：线程号

返回值：

成功：返回 0，失败返回非 0。

例：03_pthread_detach.c

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

void *thead(void *arg)
{
    int i;
    for(i=0; i<5; i++)
    {
```

```
        printf("I am runing\n");
        sleep(1);
    }
    return NULL;
}

int main(int argc, char *argv[])
{
    int ret    = 0;
    pthread_t tid1;

    ret = pthread_create(&tid1, NULL, thead, NULL);
    if(ret!=0)
        perror("pthread_create");
    pthread_detach(tid1);
    pthread_join(tid1, NULL);
    printf("after join\n");
    sleep(3);
    printf("I am leaving\n");
    return 0;
}
```

8.2.4 线程退出

在进程中我们可以调用 exit 函数或_exit 函数来结束进程，在一个线程中我们可以通过以下三种在不终止整个进程的情况下停止它的控制流。

1、线程从执行函数中返回。

2、线程调用 pthread_exit 退出线程。

3、线程可以被同一进程中的其它线程取消。

8.2.4.1 线程退出函数

```
#include <pthread.h>
void pthread_exit(void *retval);
```

功能：

退出调用线程。

参数：

retval：存储线程退出状态的指针。

注：

一个进程中的多个线程是共享该进程的数据段，因此，通常线程退出后所占用的资源并不会释放。

例：04_pthread_exit.c

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

void *thread(void *arg)
{
    int i = 0;
    while(1)
    {
        printf("I am runing\n");
        sleep(1);
        i++;
        if(i==3)
        {
            pthread_exit((void *)1);
        }
    }
    return NULL;
}

int main(int argc, char *argv[])
{
    int ret   = 0;
    pthread_t tid;
    void *value   = NULL;

    ret = pthread_create(&tid, NULL, thread, NULL);
    if(ret!=0)
        perror("pthread_create");
    pthread_join(tid, &value);
    printf("value = %p\n", (int *)value);
    return 0;
}
```

8.2.4.2 线程的取消

取消线程是指取消一个正在执行线程的操作。

#include <pthread.h>

int pthread_cancel(pthread_t thread);

功能：

取消线程。

参数：

thread:目标线程 ID。

返回值：

成功返回 0，失败返回出错编号。

pthread_cancel 函数的实质是发信号给目标线程 thread，使目标线程退出。

此函数只是发送终止信号给目标线程，不会等待取消目标线程执行完才返回。

然而发送成功并不意味着目标线程一定就会终止，线程被取消时，线程的取消属性会决定线程能否被取消以及何时被取消。

线程的取消状态

即线程能不能被取消

线程取消点

即线程被取消的地方

线程的取消类型

在线程能被取消的状态下，是立马被取消结束还是执行到取消点的时候被取消结束

线程的取消状态

在 Linux 系统下，线程默认可以被取消。编程时可以通过 pthread_setcancelstate 函数设置线程是否可以被取消。

pthread_setcancelstate(int state,int *old_state);

state：

PTHREAD_CANCEL_DISABLE：不可以被取消

PTHREAD_CANCEL_ENABLE：可以被取消。

old_state：

保存调用线程原来的可取消状态的内存地址。

例：05_pthread_setcancelstate.c 验证设置线程能\否被取消，然后看线程能不能被取消

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

void *thread_cancel(void *arg)
{
    //pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
    while(1)
    {
        printf("this is my new thread_cancel\n");
```

```
        sleep(1);
    }
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t tid1;
    int ret    = 0;
    pthread_create(&tid1, NULL, thread_cancel, NULL);
    if(ret != 0)
        perror("pthread_create");
    sleep(3);
    pthread_cancel(tid1);
    pthread_join(tid1, NULL);
    return 0;
}
```

线程的取消点

线程被取消后，该线程并不是马上终止，默认情况下线程执行到消点时才能被终止。编程时可以通过 pthread_testcancel 函数设置线程的取消点。

```
void pthread_testcancel(void);
```

当别的线程取消调用此函数的线程时候，被取消的线程执行到此函数时结束。

POSIX.1 保证线程在调用表 1、表 2 中的任何函数时，取消点都会出现。

表 1：

accept	mq_timedsend	putpmsg	sigsuspend
aio_suspend	msgrcv	pwrite	sigtimedwait
clock_nanosleep	msgsnd	read	sigwait
close	msync	readv	sigwaitinfo
connect	nanosleep	recv	sleep
creat	open	recvfrom	system
fcntl2	pause	recvmsg	tcdrain
fsync	poll	select	usleep
getmsg	pread	sem_timedwait	wait
getpmsg	pthread_cond_timedwait	sem_wait	waitid
lockf	pthread_cond_wait	send	waitpid
mq_receive	pthread_join	sendmsg	write
mq_send	pthread_testcancel	sendto	writev
mq_timedreceive	putmsg	sigpause	

表 2：

catclose	ftell	getwc	printf
catgets	ftello	getwchar	putc
catopen	ftw	getwd	putc_unlocked
closedir	fwprintf	glob	putchar
closelog	fwrite	iconv_close	putchar_unlocked
ctermid	fwscanf	iconv_open	puts
dbm_close	getc	ioctl	pututxline
dbm_delete	getc_unlocked	lseek	putwc
dbm_fetch	getchar	mkstemp	putwchar
dbm_nextkey	getchar_unlocked	nftw	readdir
dbm_open	getcwd	opendir	readdir_r
dbm_store	getdate	openlog	remove
dlclose	getgrent	pclose	rename
dlopen	getgrgid	perror	rewind
endgrent	getgrgid_r	popen	rewinddir
endhostent	getgrnam	posix_fadvise	scanf
endnetent	getgrnam_r	posix_fallocate	seekdir
endprotoent	gethostbyaddr	posix_madvise	semop
endpwent	gethostbyname	posix_spawn	setgrent
endservent	gethostent	posix_spawnp	sethostent
endutxent	gethostname	posix_trace_clear	setnetent
fclose	getlogin	posix_trace_close	setprotoent
fcntl	getlogin_r	posix_trace_create	setpwent
fflush	getnetbyaddr	posix_trace_create_withlog	setservent
fgetc	getnetbyname	posix_trace_eventtypelist_getnext_id	setutxent
fgetpos	getnetent	posix_trace_eventtypelist_rewind	strerror
fgets	getprotobyname	posix_trace_flush	syslog
fgetwc	getprotobynumber	posix_trace_get_attr	tmpfile
fgetws	getprotoent	posix_trace_get_filter	tmpnam
fopen	getpwent	posix_trace_get_status	ttyname
fprintf	getpwnam	posix_trace_getnext_event	ttyname_r
fputc	getpwnam_r	posix_trace_open	ungetc
fputs	getpwuid	posix_trace_rewind	ungetwc
fputwc	getpwuid_r	posix_trace_set_filter	unlink
fputws	gets	posix_trace_shutdown	vfprintf
fread	getservbyname	posix_trace_timedgetnext_event	vfwprintf
freopen	getservbyport	posix_typed_mem_open	vprintf
fscanf	getservent	pthread_rwlock_rdlock	vwprintf
fseek	getutxent	pthread_rwlock_timedrdlock	wprintf
fseeko	getutxid	pthread_rwlock_timedwrlock	wscanf
fsetpos	getutxline	pthread_rwlock_wrlock	

例：05_pthread_testcancel.c

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

void *thread_cancel(void *arg)
{
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    while(1)
    {
        //pthread_testcancel();
    }
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t tid1;
    int ret   = 0;
    pthread_create(&tid1, NULL, thread_cancel, NULL);
    if(ret!=0)
        perror("pthread_create");
    sleep(3);
    pthread_cancel(tid1);
    pthread_join(tid1, NULL);
    return 0;
}
```

线程的取消类型

线程被取消后，该线程并不是马上终止，默认情况下线程执行到消点时才能被终止。编程时可以通过 pthread_setcanceltype 函数设置线程是否可以立即被取消。

```
pthread_setcanceltype(int type, int *oldtype);
```

type：

PTHREAD_CANCEL_ASYNCHRONOUS：立即取消、

PTHREAD_CANCEL_DEFERRED：不立即被取消

oldtype：

保存调用线程原来的可取消类型的内存地址。

例：05_pthread_setcanceltype.c

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

void *thread_cancel(void *arg)
{
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    //pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS,NULL);
    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED,NULL);
    while(1)
    {
    }
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t tid1;
    int ret    = 0;
    pthread_create(&tid1, NULL, thread_cancel, NULL);
    if(ret!=0)
        perror("pthread_create");
    sleep(3);
    pthread_cancel(tid1);
    pthread_join(tid1, NULL);
    return 0;
}
```

8.2.4.3 线程退出清理函数

和进程的退出清理一样，线程也可以注册它退出时要调用的函数，这样的函数称为线程清理处理程序(thread cleanup handler)。

注意:

线程可以建立多个清理处理程序。

处理程序在栈中，故它们的执行顺序与它们注册时的顺序相反。

注册清理函数

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

```
void pthread_cleanup_push(void (* routine)(void *), void *arg);
```

功能:

将清除函数压栈。即注册清理函数。

参数:

routine：线程清理函数的指针。

arg：传给线程清理函数的参数。

弹出清理函数

```
#include <pthread.h>
void pthread_cleanup_pop(int execute);
```

功能：

将清除函数弹栈，即删除清理函数。

参数：

execute:线程清理函数执行标志位。

非 0，弹出清理函数，执行清理函数。

0，弹出清理函数，不执行清理函数。

当线程执行以下动作时会调用清理函数：

1、调用 pthread_exit 退出线程。

2、响应其它线程的取消请求。

3、用非零 execute 调用 pthread_cleanup_pop。

无论哪种情况 pthread_cleanup_pop 都将删除上一次 pthread_cleanup_push 调用注册的清理处理函数。

例：06_pthread_cleanup_exit.c　验证线程调用 pthread_exit 函数时，系统自动调用线程清理函数

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <string.h>
void cleanup(void *arg)
{
    printf("clean up ptr = %s\n", (char *)arg);
    free((char *)arg);
}

void *thread(void *arg)
{
    char *ptr = NULL;

    /*建立线程清理程序*/
    printf("this is new thread\n");
```

```
    ptr = (char*)malloc(100);
    pthread_cleanup_push(cleanup, (void*)(ptr));
    bzero(ptr, 100);
    strcpy(ptr, "memory from malloc");
    printf("before exit\n");
    pthread_exit(NULL);
    sleep(3);

    /*注意 push 与 pop 必须配对使用，即使 pop 执行不到*/
    printf("before pop\n");
    pthread_cleanup_pop(1);
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL);   // 创建一个线程
    pthread_join(tid,NULL);
    printf("process is dying\n");
    return 0;
}
```

例：06_pthread_cleanup_cancel.c 验证线程被取消时，系统自动调用线程清理函数

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

void cleanup(void *arg)
{
    printf("clean up ptr = %s\n", (char *)arg);
    free((char *)arg);
}

void *thread(void *arg)
{
```

```
    char *ptr = NULL;

    /*建立线程清理程序*/
    printf("this is new thread\n");
    ptr = (char*)malloc(100);
    pthread_cleanup_push(cleanup, (void*)(ptr));
    bzero(ptr, 100);
    strcpy(ptr, "memory from malloc");
    sleep(3);

    /*注意 push 与 pop 必须配对使用，即使 pop 执行不到*/
    printf("before pop\n");
    pthread_cleanup_pop(1);
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL);   // 创建一个线程
    sleep(1);
    printf("before cancel\n");
    /*子线程响应 pthread_cancel 后，会执行线程处理函数*/
    pthread_cancel(tid);
    pthread_join(tid,NULL);
    printf("process is dying\n");
    return 0;
}
```

例：06_pthread_cleanup_pop.c　　验证调用 pthread_cleanup_pop 函数时，系统自动调用线程清理函数

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

void cleanup_func1(void *arg)
{
```

```
    printf("in cleanup func1\n");
    printf("clean up ptr = %s\n", (char *)arg);
    free((char *)arg);
}

void cleanup_func2(void *arg)
{
    printf("in cleanup func2\n");
}

void *thread(void *arg)
{
    char *ptr = NULL;

    /*建立线程清理程序*/
    printf("this is new thread\n");
    ptr = (char*)malloc(100);
    pthread_cleanup_push(cleanup_func1, (void*)(ptr));
    pthread_cleanup_push(cleanup_func2, NULL);
    bzero(ptr, 100);
    strcpy(ptr, "memory from malloc");
    /*注意 push 与 pop 必须配对使用，即使 pop 执行不到*/
    printf("before pop\n");
    pthread_cleanup_pop(1);
    printf("before pop\n");
    pthread_cleanup_pop(1);
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL);   // 创建一个线程
    pthread_join(tid,NULL);
    printf("process is dying\n");
    return 0;
}
```