OPERATING SYSTEMS

Threads

Operating Systems

郑贵锋 博士
中山大学计算机学院
zhenggf@mail.sysu.edu.cn
https://gitee.com/code_sysu

中山大學 SUN YAT-SEN UNIVERSITY

Overview

- **目录**
  - 概述
  - 多核编程
  - 用户和内核级线程
  - 多线程模型
  - **线程库**
    - POSIX**线程**Pthreads
  - **隐式线程化实现**
    - **线程池**
    - OpenMP
  - **线程问题**
    - fork()**和**exec()**系统调用的语义**
    - **信号处理**
    - **线程取消**
    - 线程本地存储
    - 调度程序激活
  - Linux clone()
  - 示例：Windows线程

Thread Libraries

- **线程库**
  - 线程库为程序员提供了创建和管理线程的API
  - 有两种实现线程库的主要方法。
    - 完全在用户空间中提供一个库，没有内核支持。
      - 库的所有代码和数据结构都存在于用户空间中。
      - 调用库中的函数会导致用户空间中的本地函数调用，而不是系统调用。
    - 实现操作系统直接支持的内核级库。
      - 库的代码和数据结构存在于内核空间中。
      - 在库的API中调用函数通常会导致对内核的系统调用。

Thread Libraries

- **线程库**
  - 目前使用的主要线程库有三个：
    - POSIX线程
      - Pthreads是POSIX标准的线程扩展，可以作为用户级库或内核级库提供。
      - 在UNIX和Linux系统上使用。
    - Windows线程库
      - Windows系统上可用的内核级库。
    - Java线程API
      - 它允许在Java程序中直接创建和管理线程。
      - 在大多数情况下，JVM在主机操作系统上运行，Java线程API通常使用主机系统上可用的线程库实现。
      - 这意味着在Windows系统上，Java线程通常使用Windows API实现。

Thread Libraries

- **线程库**
  - 创建多线程的两种常用策略：异步线程和同步线程
    - 异步线程
      - 一旦父线程创建了子线程，父线程将恢复执行，以便父线程和子线程同时执行。
      - 每个线程独立于其他线程运行，父线程不需要知道其子线程何时终止。
      - 线程之间通常很少有数据共享。
    - 同步线程
      - 父线程创建一个或多个子线程，然后等待所有子线程终止之后才恢复运行，这就是所谓的*fork-join*(分支聚合)策略.
      - 父线程创建的线程同时执行工作。一旦每个线程完成其工作，它就会终止并与其父线程连接。只有在所有子级都加入后，父级才能恢复执行。
      - 涉及线程之间的显著数据共享。

Thread Libraries

- **线程库**
  - POSIX线程
    - *POSIX Pthreads*是指POSIX标准（IEEE 1003.1c），该标准定义了用于创建和同步线程的API。它为ULT或KLT提供支持。
    - Pthreads是线程行为的规范。操作系统设计者可以按照他们希望的任何方式实现规范。
    - 许多系统都实现了Pthreads规范。
      - UNIX类型的系统，包括Linux、MacOS X和Solaris

线程调用	描述
pthread_create	创建一个新线程
pthread_exit	终止调用的线程
pthread_join	等待特定线程退出
pthread_yield	释放CPU，让另一个线程运行
pthread_attr_init	创建并初始化线程的属性结构
pthread_attr_destroy	删除线程的属性结构

一些Pthreads函数调用

Thread Libraries

- **线程库**
  - POSIX线程
    - Windows原生不支持Pthreads.
      - 有一些第三方实现可用
    - 下一张幻灯片中显示的C程序演示了基本的Pthreads API，用于构造一个多线程程序，该程序在单独的线程中计算一个非负整数的和:

$$sum = \sum_{i=0}^{N} i$$

Thread Libraries

- **线程库**
  - Pthreads–创建并连接
    - 算法.13-1-pthread-create.c (1)

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

```
/* gcc -lpthread | -pthread */
int sum; /* shared by threads */
static void *runner(void *); /* thread function */
int main(int argc, char *argv[])
{
    if(argc < 2) {
        printf("usage: ./a.out <positive integer value>\n");
        return -1;
    }
    pthread_t ptid; /* thread identifier */
    pthread_attr_t attr; /* thread attributes structure */
    pthread_attr_init(&attr); /* set the default attributes */
      /* create the thread - runner with argv[1] */
    ret = pthread_create(&ptid, &attr, &runner, argv[1]);
    if (ret != 0) {
        perror("pthread_create()");
        return 1;
    }
    ret = pthread_join(ptid, NULL); /* join() waiting until thread tid returns */
    if (ret != 0) {
        perror("pthread_join()");
        return 1;
    }
    printf("sum = %d\n", sum);
    return 0;
}
```

创建运行线程runner，传递命令行参数；
join()阻塞等待线程结束返回；
共享的sum全局变量被线程更新；

Thread Libraries

- **线程库**
  - Pthreads–创建并连接
    - **算法**. 13-1-pthread-create.c (2)

```
/* The thread will begin control in this function */
static void *runner(void *param)
{
    int i, upper;

    upper = atoi(param);
    sum = 0;
    for (I = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
```

工作线程的runner函数：
1. 获得传入参数；
2. 计算和，更新共享的sum全局变量的值；
3. exit()结束返回；

Thread Libraries

- **线程库**
  - Pthreads–创建并连接
    - **算法**. 13-1-pthread-create.c (2)

```
/* The thread will begin control in this function */
static void *runner(void *param)
{
    int i, upper;

    upper = atoi(param);
    sum = 0;
```

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.13-1-pthread-create.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out
usage: a.out <positive integer value>
isscgy@ubuntu:/mnt/os-2020$ ./a.out 10
sum = 55
isscgy@ubuntu:/mnt/os-2020$ ./a.out 100
sum = 5050
isscgy@ubuntu:/mnt/os-2020$ ./a.out -10
sum = 0
isscgy@ubuntu:/mnt/os-2020$ ./a.out asd
sum = 0
isscgy@ubuntu:/mnt/os-2020$
```

Thread Libraries

- **线程库**
  - Pthreads–创建并连接
    - **算法**. 13-1-pthread-create.c
      - pthread.h 任何Pthreads程序都必须包含这个头文件。
      - 当程序开始时，一个控制线程从main()开始。初始化之后，main()创建第二个线程，该线程开始在runner()函数中进行控制。两个线程共享全局数据sum。
      - pthread_t tid声明我们将创建的线程的标识符tid。
      - pthread_attr_t attr声明线程的属性，由pthread_attr_init(&attr)设置。在没有显式设置的情况下将使用默认属性。
      - pthread_create() 创建一个单独的线程，使用4个参数：
        - 新线程生成的标识符将传递给tid
        - 新线程要使用属性attr
        - 新线程要执行函数runner()
        - 新线程执行函数时的参数来自命令行提供的整数参数argv[1]

Thread Libraries

- **线程库**
  - Pthreads–创建并连接
    - **算法**. 13-1-pthread-create.c
      - 现在有两个线程：main()中的父线程和runner()中执行求和的子线程。
      - 程序遵循fork-join同步策略：创建求和线程后，父线程将通过调用pthread_join()函数等待它终止。
      - 求和线程将在调用函数pthread_exit()时终止。一旦求和线程返回，父线程将恢复，输出共享数据sum的值。
      - 这个示例程序只创建一个线程。使用pthread_join()函数等待多个线程的一个简单方法是使用for循环：

```
#define NUM_THREADS 10
pthread_t workers[NUM_THREADS];
    … …
for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
```

Thread Libraries

- **线程库**
  - Pthreads–创建并连接
    - **算法**. 13-1-pthread-create.c
      - void pthread_exit(void *retval); int pthread_join(tid1, &tret);
      - 如果pthread_join()的第二个参数不为NULL，将得到pthread_exit()退出时传回的返回值*retval

```
/* alg.13-1-pthread-create-1-1.c */
        … …
int main(int argc, char *argv[])
{
        … …
    pthread_create(&ptid, &attr, &runner, argv[1]);
        … …
    int *retptr;
    pthread_join(ptid, (void **)&retptr);
    printf("runner returned val = %d\n", *retptr);
        … …
}

static void *runner(void *param)
{
        … …
    int *retptr = (int *)malloc(sizeof(int)); /* allocated in process space */
    *retptr = 16;
    pthread_exit((void *)retptr);
}
```

Thread Libraries

- **线程库**
  - Pthreads–创建并连接
    - **算法**. 13-1-pthread-create.c
      - void pthread_exit(void *retval); int pthread_join(tid1, &tret);
      - 如果pthread_join()的第二个参数不为NULL，将得到pthread_exit()退出时传回的返回值*retval

```
/* alg.13-1-pthread-create-1-2.c */
int sum;
        … …
int main(int argc, char *argv[])
{
        … …
    pthread_create(&ptid, &attr, &runner, argv[1]);
        … …
    int *retptr;
    pthread_join(ptid, (void **)&retptr);
    printf("runner returned val = %d\n", *retptr);
        … …
}

static void *runner(void *param)
{
        … …
    pthread_exit((void *)&sum);  /* 子线程结束时也返回全局变量sum的地址 */
}                                /* 也可以写成 return (void *)&sum;    */
```

Thread Libraries

- **线程库**
  - Pthreads–创建并连接
    - 算法. 13-1-pthread-create.c
      - void pthread_exit(void *retval); int pthread_join(tid1, &tret);
      - 如果pthread_join()的第二个参数不为NULL，将得到pthread_exit()退出时传回的返回值*retval

```
/* alg.13-1-pthread-create-1-3.c */
/* 不定义全局变量sum，而是使用局部变量 */
int main(void)
{
    int sum;
    int *retptr = &sum;     … …

    pthread_create(&ptid, &attr, &runner, &sum);
        … …
    pthread_join(ptid, (void **)&sun);
    printf("runner returned val = %d\n", *retptr);
        … …
}

static void *runner(void *param)
{
    int *sum = (int *)param;
        … …
    pthread_exit((void *)sum); /* 返回的地址在进程的堆栈段 */
}
```

Thread Libraries

- **线程库**
  - Pthreads–创建并连接
    - 算法. 13-1-pthread-create.c
      - void pthread_exit(void *retval); int pthread_join(tid1, &tret);
      - 如果pthread_join()的第二个参数不为NULL，将得到pthread_exit()退出时传回的返回值*retval

```
/* alg.13-1-pthread-create-2.c */
        … …
#include <string.h>
int main(int argc, char *argv[])
{       … …
    pthread_create(&tid, &attr, &runner, argv[1]);
        … …
    char *retptr;
    pthread_join(tid, (void **)&retptr);
    printf("runner returned val = %s\n", retptr);
        … …
}

static void *runner(void *param)
{       … …
    char mag[] = "Hello world!";
    char *retptr = (char *)malloc((strlen(msg) + 1)*sizeof(char));
    strcpy(retptr, msg);
    pthread_exit((void *)retptr); /* 在子进程动态分配内存空间 */
}
```

线程库

Pthreads–创建并连接

算法. 13-1-pthread-create-3.c (1)

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

static void *ftn(void *arg)
{
    int *numptr = (int *)arg;
    int num = *numptr;
    char *retval = (char *)malloc(80*sizeof(char));  /* 在进程的堆中分配 */

    sprintf(retval, "This is thread-%d, ptid = %lu", num, pthread_self( ));
    printf("%s\n", retval);

    pthread_exit((void *)retval); /* or return (void *)retval; */
}

int main(int argc, char *argv[])
{
    int max_num = 5;
    int i, ret;

    printf("Usage: ./a.out total_thread_num\n");
    if(argc > 1)
        max_num = atoi(argv[1]);
    printf("main(): pid = %d, ptid = %lu.\n", getpid( ), pthread_self( ));
```


线程库

Pthreads–创建并连接

算法. 13-1-pthread-create-3.c (2)

把i变量的地址作为参数

```
    pthread_t ptid[max_num];
    for(i = 0; i < max_num; i++) {
        ret = pthread_create(&ptid[i], NULL, ftn, (void *)&i);
        if(ret != 0) {
            fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
            exit(1);
        }
    }

    for(i = 0; i < max_num; i++) {
        char *retptr;  /* retptr pointing to address allocated by ftn() */
        ret = pthread_join(ptid[i], (void **)&retptr);
        if(ret!=0) {
            fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
            exit(1);
        }
        printf("thread-%d: retval = %s\n", i, retptr);
        free(retptr);
        retptr = NULL;  /* preventing ghost pointer */
    }

    return 1;
}
```

Thread Libraries

- **线程库**
  - Pthreads–创建并连接
    - **算法**. 13-1-pthread-create-3.c

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.13-1-pthread-create-3.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out 5
Usage: ./a.out total_thread_num
main(): pid = 18960, ptid = 140253726336832.
This is thread-2, ptid = 140253717825280
This is thread-3, ptid = 140253709432576
This is thread-4, ptid = 140253701039872
This is thread-0, ptid = 140253684254464
This is thread-0, ptid = 140253692647168
thread-0: retval = This is thread-2, ptid = 140253717825280
thread-1: retval = This is thread-3, ptid = 140253709432576
thread-2: retval = This is thread-4, ptid = 140253701039872
thread-3: retval = This is thread-0, ptid = 140253692647168
thread-4: retval = This is thread-0, ptid = 140253684254464
isscgy@ubuntu:/mnt/os-2020$
```

i 与 ptid[i] 不一致！
- 线程读取进程变量i的变化的值；
- 线程结束不是按顺序的；

Thread Libraries

- **线程库**
  - Pthreads–创建并连接
    - **算法**. 13-1-pthread-create-3-1.c

i与 ptid[i]不一致问题的解决方法一：
创建线程后等1秒

```
    pthread_t ptid[max_num];
    for(i = 0; i < max_num; i++) {
        ret = pthread_create(&ptid[i], NULL, ftn, (void *)&i);
        if(ret != 0) {
            fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
            exit(1);
        }
        sleep(1);
    }

    for(i = 0; i < max_num; i++) {
        char *retptr;  /* retptr pointing to address allocated by ftn() */
        ret = pthread_join(ptid[i], (void **)&retptr);
        if(ret!=0) {
            fprintf(stderr, "pthread_join error: %s\n", strerror(ret));
            exit(1);
        }
        printf("thread-%d: retval = %s\n", i, retptr);
        free(retptr);
        retptr = NULL;  /* preventing ghost pointer */
    }

    return 1;
}
```

Thread Libraries

- **线程库**
  - Pthreads–创建并连接
    - **算法**. 13-1-pthread-create-3-1.c

i与 ptid[i]不一致问题的解决方法一：
创建线程后等待1秒

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.13-1-pthread-create-3-1.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out 5
Usage: ./a.out total_thread_num
main(): pid = 18977, ptid = 139918462719808.
This is thread-0, ptid = 139918454208256
This is thread-1, ptid = 139918443620096
This is thread-2, ptid = 139918435227392
This is thread-3, ptid = 139918357559040
This is thread-4, ptid = 139918349166336
thread-0: retval = This is thread-0, ptid = 139918454208256
thread-1: retval = This is thread-1, ptid = 139918443620096
thread-2: retval = This is thread-2, ptid = 139918435227392
thread-3: retval = This is thread-3, ptid = 139918357559040
thread-4: retval = This is thread-4, ptid = 139918349166336
isscgy@ubuntu:/mnt/os-2020$
```

```
    }

    return 1;
}
```

Thread Libraries

- **线程库**
  - Pthreads–创建并连接
    - **算法**. 13-1-pthread-create-4.c

i与 ptid[i]不一致问题的解决方法二：
用thread_num数组为线程准备独立的编号

```
int main(int argc, char *argv[])
{
    int max_num = 5;
    int i, ret;

    printf("Usage: ./a.out total_thread_num\n");
    if(argc > 1) {
        max_num = atoi(argv[1]);
    }

    int thread_num[max_num];
    for (i = 0; i < max_num; i++) {
        thread_num[i] = i;
    }

    printf("main(): pid = %d, ptid = %lu.\n", getpid( ), pthread_self( ));

    pthread_t ptid[max_num];
    for(i = 0; i < max_num; i++) {
        ret = pthread_create(&ptid[i], NULL, ftn, (void *)&thread_num[i]);
        if(ret != 0) {
            fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
            exit(1);
        }
    }
```

Thread Libraries

- **线程库**
  - Pthreads–创建并连接
    - **算法**. 13-1-pthread-create-4.c

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.13-1-pthread-create-4.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out 5
Usage: ./a.out total_thread_num
main(): pid = 19020, ptid = 140096962860864.
This is thread-0, ptid = 140096954349312
This is thread-1, ptid = 140096945956608
This is thread-3, ptid = 140096858744576
This is thread-4, ptid = 140096850351872
This is thread-2, ptid = 140096867137280
thread-0: retval = This is thread-0, ptid = 140096954349312
thread-1: retval = This is thread-1, ptid = 140096945956608
thread-2: retval = This is thread-2, ptid = 140096867137280
thread-3: retval = This is thread-3, ptid = 140096858744576
thread-4: retval = This is thread-4, ptid = 140096850351872
isscgy@ubuntu:/mnt/os-2020$
```

i与 ptid[i]不一致问题的解决方法二：用thread_num数组为线程准备独立的编号

```
            fprintf(stderr, "pthread_create error: %s\n", strerror(ret));
            exit(1);
        }
    }
```

Thread Libraries

- **线程库**
  - Pthreads–共享内存
    - **算法**. 13-2-pthread-shm.c (1)

```
struct msg_stru {
    char msg1[MSG_SIZE], msg2[MSG_SIZE], msg3[MSG_SIZE];
}; /* 全局结构 */

static void *runner1(void *);
static void *runner2(void *);

int main(void)
{
    pthread_t tid1, tid2;
    pthread_attr_t attr = {0};
    struct msg_stru msg; /* 保存在主线程的堆栈 */

    sprintf(msg.msg1, "message 1 by parent");
    sprintf(msg.msg2, "message 2 by parent");
    sprintf(msg.msg3, "message 3 by parent");
    printf("\nparent say:\n%s\n%s\n%s\n", msg.msg1, msg.msg2, msg.msg3);

    pthread_attr_init(&attr);
    if(pthread_create(&tid1, &attr, &runner1, (void *)&msg) != 0) {
        perror("pthread_create()"); return 1;
    }
    if(pthread_create(&tid2, &attr, &runner2, (void *)&msg) != 0) {
        perror("pthread_create()"); return 1;
    }
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <pthread.h>
#define MSG_SIZE 1024
```

Thread Libraries

- **线程库**
  - Pthreads–**共享内存**
    - **算法**. 13-2-pthread-shm.c (2)

```
    if(pthread_join(tid1, NULL) != 0) {
        perror("pthread_join()"); return 1;
    }
    if(pthread_join(tid2, NULL) != 0) {
        perror("pthread_join()"); return 1;
    }

    printf("\nparent say:\n%s\n%s\n%s\n", msg.msg1, msg.msg2, msg.msg3);
    return 0;
}

static void *runner1(void *param)
{
    struct msg_stru *ptr = (struct msg_stru *)param;
    sprintf(ptr->msg1, "message 1 changed by child1");
    pthread_exit(0);
}

static void *runner2(void *param)
{
    struct msg_stru *ptr = (struct msg_stru *)param;
    sprintf(ptr->msg2, "message 2 changed by child2");
    pthread_exit(0);
}
```

Thread Libraries

- **线程库**
  - Pthreads–**共享内存**
    - **算法**. 13-2-pthread-shm.c (2)

```
    if (pthread_join(tid1, NULL) != 0) {
```

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.13-2-pthread-shm.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out

parent say:
message 1 by parent
message 2 by parent
message 3 by parent

parent say:
message 1 changed by child1
message 2 changed by child2
message 3 by parent
isscgy@ubuntu:/mnt/os-2020$
```

```
{
    struct msg_stru *ptr = (struct msg_stru *)param;
    sprintf(ptr->msg2, "message 2 changed by child2");
    pthread_exit(0);
}
```

Thread Libraries

线程库

- Pthreads – 测试pthread_attr_setstack设置线程栈
  - 算法. 13-3-pthread-stack.c (1)

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <malloc.h>
#define STACK_SIZE (524288-10000)*4096 /* 接近 2G: 2^19 = 524288 */

int i = 0;
static void *test(void *arg)
{
    char buffer[1024]; /* 1KiB 保存在线程栈 */
    if(i > 5 && i < 1965030)
       printf("\b\b\b\b\b\b\b\b%8d", i);
    else
       printf("\niteration = %8d", i);
    i++;
    test(arg); /* 递归调用直至段错误 */
}
```

Thread Libraries

线程库

- Pthreads – 测试pthread_attr_setstack设置线程栈
  - 算法. 13-3-pthread-stack.c (1)

```
int main(void)
{
    int ret;
    pthread_t ptid;
    pthread_attr_t tattr = {0};
    char *stackptr = malloc(STACK_SIZE);
    if(!stackptr) {
        perror("malloc()");
        return 1;
    }
    pthread_attr_init(&tattr);
    pthread_attr_setstack(&tattr, stackptr, STACK_SIZE);
    ret = pthread_create(&ptid, &tattr, &test, NULL);
    if(ret) {
        perror("pthread_create()");
        return 1;
    }
    sleep(1);
    pthread_join(ptid, NULL);

    return 0;
}
```

Thread Libraries

- **线程库**
  - Pthreads – 测试pthread_attr_setstack设置线程栈
    - 算法. 13-3-pthread-stack.c (1)

```
char *stackptr = malloc(STACK_SIZE);
if (!stackptr) {
    perror("malloc()");
```

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.13-3-pthread-stack.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out

iteration =       0
iteration =       1
iteration =       2
iteration =       3
iteration =       4
iteration =  1965029
iteration =  1965030
iteration =  1965031
iteration =  1965032
Segmentation fault (core dumped)
isscgy@ubuntu:/mnt/os-2020$
```

STACK_SIZE是514288*4096，每次迭代用了1K，剩余94330880字节：
514288*4096-1965032*1024 = 94330880
这些内存是系统递归迭代消耗的成本，平均每次迭代需要使用大约48字节:
94330880/1965032 = 48 (bytes)

Implicit Threading

- **隐式线程化实现**
  - *隐式线程*是一种为多线程应用程序的设计提供更好支持的策略。线程的创建和管理从应用程序开发人员转移到编译器和运行时库。
  - 设计多线程程序有三种可选方法，可以通过隐式线程利用多核处理器。
    - 线程池
    - OpenMP
    - Grand Central Dispatch（苹果iOS）

Implicit Threading

- **隐式线程化实现**
  - 线程池Thread Pools
    - 问题:
      - 无限线程可能耗尽系统资源，如CPU时间或内存。
    - 解决方案
      - 在进程启动时创建一些线程，并将它们放入线程池中，它们坐在那里等待工作。
      - 当服务器收到请求时，如果有线程可用，它会从该池中唤醒线程，并将服务请求传递给它。
      - 一旦线程完成其服务，它将返回池并等待更多的工作。如果池中不包含可用线程，服务器将等待一个线程空闲。
    - 实例
      - IA-32中每个进程的最大线程数。
        - 在3G地址空间中，这个数字大约为300，每个线程的默认堆栈大小为*10M*
        - 可以创建一个少于255个线程的线程池

Implicit Threading

- **隐式线程化实现**
  - 线程池
    - 线程池的好处
      - 使用现有线程为请求提供服务通常比等待创建新线程要快一些。
      - 线程池限制任何一点上存在的线程数。对于不能支持大量并发线程的系统，这一点尤为重要。
      - 将要执行的任务与创建任务的机制分离，允许我们使用不同的策略来运行任务。
    - 线程池的大小
      - 池中线程的数量可以根据系统中CPU的数量、物理内存的数量以及并发客户端请求的预期数量等因素进行启发式设置
      - 更复杂的线程池架构（如苹果的Grand Central Dispatch）可以根据使用模式动态调整池中的线程数量。

Implicit Threading

隐式线程化实现

- OpenMP
  - OpenMP是一组编译器指令，以及用C、C++或FORTRAN编写的程序API，它提供了对共享内存环境中并行编程的支持。
  - OpenMP将并行区域标识为可以并行运行的代码块。
    - 应用程序开发人员在并行区域向代码中插入编译器指令，这些指令指示OpenMP运行时库并行执行该区域。
  - 下面的C程序演示了包含printf()语句的并行区域上方的编译器指令。
    - 当OpenMP遇到指令时

      ```
      #pragma omp parallel
      ```

      - 它创建的线程数量与系统中处理内核的数量相同（例如，Intel CPU每内核两个线程）。所有线程同时执行并行区域。当每个线程退出并行区域时，将被终止执行

Implicit Threading

隐式线程化实现

- OpenMP
  - 算法. 13-4-openmp-demo.c

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

```
/* 编译命令: gcc -fopenmp */
#define __NR_gettid 186 /* 186 for x86-64; 224 for i386-32 */
#define gettid() syscall(__NR_gettid)

int main()
{
    #pragma omp parallel
    {
        printf("region 1. pid = %d, tid = %ld\n", getpid(), gettid());
    }
    #pragma omp parallel num_threads(2)
    {
        printf("region 2, num_thread(2). pid = %d, tid = %ld\n", getpid(), gettid());
    }
    #pragma omp parallel num_threads(4)
    {
        printf("region 3, num_thread(4). pid = %d, tid = %ld\n", getpid(), gettid());
    }
    #pragma omp parallel num_threads(6)
    {
        printf("region 4, num_thread(6). pid = %d, tid = %ld\n", getpid(), gettid());
    }

    return 0;
}
```

Implicit Threading

- **隐式线程化实现**
  - OpenMP
    - 算法. 13-4-openmp-demo.c

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

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.13-4-openmp-demo.c -fopenmp
isscgy@ubuntu:/mnt/os-2020$ ./a.out
parallel region 1. pid = 19375, tid = 19375
parallel region 1. pid = 19375, tid = 19378
parallel region 1. pid = 19375, tid = 19377
parallel region 1. pid = 19375, tid = 19376
parallel region 2 with num_thread(2). pid = 19375, tid = 19376
parallel region 2 with num_thread(2). pid = 19375, tid = 19375
parallel region 3 with num_thread(4). pid = 19375, tid = 19375
parallel region 3 with num_thread(4). pid = 19375, tid = 19379
parallel region 3 with num_thread(4). pid = 19375, tid = 19376
parallel region 3 with num_thread(4). pid = 19375, tid = 19380
parallel region 4 with num_thread(6). pid = 19375, tid = 19379
parallel region 4 with num_thread(6). pid = 19375, tid = 19380
parallel region 4 with num_thread(6). pid = 19375, tid = 19376
parallel region 4 with num_thread(6). pid = 19375, tid = 19375
parallel region 4 with num_thread(6). pid = 19375, tid = 19382
parallel region 4 with num_thread(6). pid = 19375, tid = 19381
isscgy@ubuntu:/mnt/os-2020$
```

tid 19376 线程被重用

Implicit Threading

- **隐式线程化实现**
  - OpenMP
    - OpenMP为并行运行代码区域提供了几个附加指令，包括并行化循环。
      - 假设我们有两个大小为N的数组a和b。我们希望对它们的内容求和，并将结果放入数组c中。我们可以使用以下代码段并行运行此任务，其中包含用于循环并行化的编译器指令

```
#pragma omp parallel for
/* 将for循环中包含的工作在它所创建的线程中分配 */
for (i = 0; i < N; i++) {
    c[i] = a[i] + b[i];
}
```

隐式线程化实现

- OpenMP
  - 算法. 13-5-openmp-matrixadd.c (1)

```
/* compliling: gcc -fopenmp */
/* omp适用于矩阵相加：当维度od=12200，使用4线程时，系统利用率可能有30%的改进 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <ctype.h>
#include <omp.h>
#define MAX_N 12288

int a[MAX_N][MAX_N], b[MAX_N][MAX_N];
int ans[MAX_N][MAX_N];
int od = 10;

void matrixadd(void)
{
    for(int i = 0; i < od; i++){
        for(int j = 0; j < od; j++){
            ans[i][j] = a[i][j] + b[j][j];
        }
    }

    return;
}
```


隐式线程化实现

- OpenMP
  - 算法. 13-5-openmp-matrixadd.c (2)

```
int main(int argc, void *argv[])
{
    int i, iteration;

    if(argc > 1)
        od = atoi(argv[1]);
    if (od > MAX_N || od < 0)
        return 1;
    i = MAX_N;
    printf("MAX_N = %d, od = %d\n", i, od);
    for(int i = 0; i < od; i++)
        for(int j = 0; j < od; j++)
            a[i][j] = 20;
    for(int i = 0; i < od; i++)
        for(int j = 0; j < od; j++)
            b[i][j] = 30;

      /* 不使用omp */
    long start_us, end_us;
    struct timeval t;
    gettimeofday(&t, 0);
    start_us = (long)(t.tv_sec * 1000 * 1000) + t.tv_usec;
    matrixadd();
    gettimeofday(&t, 0);
    end_us = (long)(t.tv_sec * 1000 * 1000) + t.tv_usec;
    printf("Overhead time usec = %ld, with no omp\n", end_us-start_us);
```

Implicit Threading

- **隐式线程化实现**
  - OpenMP
    - **算法**. 13-5-openmp-matrixadd.c (3)

```
  /* 使用 2 线程 */
gettimeofday(&t, 0);
start_us = (long)(t.tv_sec * 1000 * 1000) + t.tv_usec;
#pragma omp parallel num_threads(2)
{
    matrixadd();
}
gettimeofday(&t, 0);
end_us = (long)(t.tv_sec * 1000 * 1000) + t.tv_usec;
printf("Overhead time usec = %ld, omp thread = 2\n", end_us-start_us);

  /* 使用 4 线程 */
gettimeofday(&t, 0);
start_us = (long)(t.tv_sec * 1000 * 1000) + t.tv_usec;
#pragma omp parallel num_threads(4)
{
    matrixadd();
}
gettimeofday(&t, 0);
end_us = (long)(t.tv_sec * 1000 * 1000) + t.tv_usec;
printf("Overhead time usec = %ld, omp thread = 4\n", end_us-start_us);

    return 0;
}
```

Implicit Threading

- **隐式线程化实现**
  - OpenMP
    - **算法**. 13-5-openmp-matrixadd.c (3)

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.13-5-openmp-matrixadd.c -fopenmp
isscgy@ubuntu:/mnt/os-2020$ ./a.out 10
MAX_N = 12288, od = 10
Overhead time usec = 17, with no omp
Overhead time usec = 96, omp thread = 2
Overhead time usec = 2779, omp thread = 4
isscgy@ubuntu:/mnt/os-2020$ ./a.out 1000
MAX_N = 12288, od = 1000
Overhead time usec = 15797, with no omp
Overhead time usec = 14968, omp thread = 2
Overhead time usec = 13119, omp thread = 4
isscgy@ubuntu:/mnt/os-2020$ ./a.out 12200
MAX_N = 12288, od = 12200
Overhead time usec = 6487102, with no omp
Overhead time usec = 4929145, omp thread = 2
Overhead time usec = 3538193, omp thread = 4
isscgy@ubuntu:/mnt/os-2020$
```

Threading Issues

- **线程问题**
  - 在设计多线程程序时需要考虑的一些问题。
    - fork()和exec()系统调用的语义
    - 信号处理
    - 线程取消
      - 异步的或延迟的。
    - 线程本地存储
    - 调度程序激活

Threading Issues

- **fork()和exec()**
  - fork()系统调用用于创建单独的重复进程。但是fork()和exec()系统调用的语义在多线程程序中会发生变化。
    - 如果程序中的一个线程调用fork()，则新进程可能会失败
      - 复制所有线程，或
      - 只有调用fork()系统调用的线程（在Ubuntu中）。
        - 它可能有很高的风险。
    - 某些UNIX系统有两个版本的fork()。
  - exec()系统调用的工作方式通常是，如果某个线程调用exec()系统调用，则exec()参数中指定的程序将替换调用进程，包括其所有线程
    - 如果在分叉之后立即调用exec()，分叉的进程只需要复制调用线程。
      - 不需要复制所有线程，因为exec()参数中指定的程序将替换调用进程
    - 否则，分支进程在分支之后不会调用exec()，它应该复制调用进程的所有线程。
  - 建议：避免在多线程中使用fork()

Threading Issues

- **fork()和exec()**
  - fork()和多线程
    - **算法**. 13-6-fork-pthread-demo1.c (1)

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/wait.h>

int i = 0;

static void *thread_worker(void *args)
{
    while (1) {
        printf("%d\n", i);
          /* 每秒打印输出0, 由父进程main()函数创建的thread_worker线程 */
        sleep(1);
    }

    pthread_exit(0);
}
```

Threading Issues

- **fork()和exec()**
  - fork()和多线程
    - **算法**. 13-6-fork-pthread-demo1.c (2)

```
int main(void)
{
    pthread_t ptid;
    pthread_create(&ptid, NULL, &thread_worker, NULL);

    pid_t pid = fork(); /* 子进程只复制父进程的主线程, 不复制thread_worker */
    if(pid == 0){ /* 子进程 */
        i = 1;
        printf("In child\n");
        system("ps -l -T");
        while (1) ;
        exit (0);
    }
    /* 父进程 */
    wait(&pid);
    printf("In parent\n");
    system("ps -l -T");
    while (1) ;

    return 0;
}
```

Threading Issues

- **fork()和exec()**

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.13-6-fork-pthread-demo1.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out
in child
0
F S   UID     PID    SPID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000    1856    1856    1846  0  80   0 -  6124 wait   pts/0    00:00:04 bash
0 S  1000   19526   19526    1856  0  80   0 - 20107 wait   pts/0    00:00:00 a.out
1 S  1000   19526   19527    1856  0  80   0 - 20107 hrtime pts/0    00:00:00 a.out
1 S  1000   19528   19528   19526  0  80   0 -  3723 wait   pts/0    00:00:00 a.out
0 S  1000   19529   19529   19528  0  80   0 -  1158 wait   pts/0    00:00:00 sh
0 R  1000   19530   19530   19529  0  80   0 -  7667 -      pts/0    00:00:00 ps
in parent
F S   UID     PID    SPID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000    1856    1856    1846  0  80   0 -  6124 wait   pts/0    00:00:04 bash
0 S  1000   19526   19526    1856  0  80   0 - 20107 wait   pts/0    00:00:00 a.out
1 S  1000   19526   19527    1856  0  80   0 - 20107 hrtime pts/0    00:00:00 a.out
0 S  1000   19531   19531   19526  0  80   0 -  1158 wait   pts/0    00:00:00 sh
0 R  1000   19532   19532   19531  0  80   0 -  7667 -      pts/0    00:00:00 ps
0
0
0
0
^C
isscgy@ubuntu:/mnt/os-2020$
```

只输出0, 由父进程创建的线程thread_worker输出

pid=19526, spid=19526:父进程main()线程
pid=19526, spid=19527:父进程thread_worker线程
pid=19528, spid=19528:子进程（没有其他线程）

```
}
```

Threading Issues

- **fork()和exec()**
  - fork()和多线程
    - **算法**. 13-7-fork-pthread-demo2.c (1)

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

int i = 0;
static void *thread_worker(void *args)
{
    pid_t pid = fork(); /* 分支子进程将thread_worker作为main线程.
                           在同步或信号处理过程中将导致异常行为 */
    if(pid < 0 )
        return (void *)EXIT_FAILURE;
    if(pid == 0) { /* 子进程 */
        i = 1;
        printf("In thread_worker's forked child\n");
        system("ps -l -T");
    }
    /* 子进程未exit，因此父子进程都执行下面的代码 */
    sleep(2);
    while (1) {
        printf("%d\n", i);
        /* 父进程的thread_worker将输出‘0’; 子进程则输出‘1’ */
        sleep(2);
    }
    pthread_exit(0);
}
```

Threading Issues

- **fork()和exec()**
  - fork()和多线程
    - **算法**. 13-7-fork-pthread-demo2.c (2)

```
int main(void)
{
    pthread_t tid;
    pthread_create(&tid, NULL, &thread_worker, NULL);

    sleep(2) ;
    printf("In start main()\n");
    system("ps -l -T");

    while (1) ;

    pthread_join(tid, NULL);

    return EXIT_SUCCESS;
}
```

Threading Issues

- **fork()和exec()**
  - fork()和多线程
    - **算法**. 13-7-fork-pthread-demo2.c (2)

```
isscgy@ubuntu:/mnt/os-2020$ gcc alg.13-7-fork-pthread-demo2.c -pthread
isscgy@ubuntu:/mnt/os-2020$ ./a.out
in thread_worker's forked child
0 S  1000  19605  19605   1856  0  80   0 -  3723 hrtime pts/0    00:00:00 a.out
1 S  1000  19605  19606   1856  0  80   0 -  3723 hrtime pts/0    00:00:00 a.out
1 S  1000  19607  19607  19605  0  80   0 -  3723 wait   pts/0    00:00:00 a.out
in start main()
0
1
0 S  1000  19605  19605   1856  0  80   0 -  3723 wait   pts/0    00:00:00 a.out
1 S  1000  19605  19606   1856  0  80   0 -  3723 hrtime pts/0    00:00:00 a.out
1 S  1000  19607  19607  19605  0  80   0 -  3723 hrtime pts/0    00:00:00 a.out
0
1
0
1
0
1
^C
isscgy@ubuntu:/mnt/os-2020$
```

pid=19605, spid=19605: 父进程main()线程
pid=19605, spid=19605: 父进程thread_worker线程
pid=19607, spid=19607: 子进程thread_worker线程

父进程thread_worker打印0
而子进程打印1

Threading Issues

- **fork()和exec()**
  - fork()和多线程
    - **算法**. 13-7-fork-pthread-demo2.c (2)

```
int main(void)
{
    pthread_t ptid;
    pthread_create(&ptid, NULL, &thread_worker, NULL);

    sleep(2) ;
    printf("in start main()\n");
    system("ps -l -T");

    return 1; /* 会发生什么？ 必须pkill分支的子进程 */

    while (1) ;

    pthread_join(tid, NULL);

    return EXIT_SUCCESS;
}
```

Threading Issues

- **fork()和exec()**
  - fork()和多线程
    - **算法**. 13-7-fork-pthread-demo2.c (2)

```
isscgy@ubuntu:/mnt/os-2020$ ./a.out
in thread_worker's forked child
0 S  1000  19647  19647   1856  0  80   0 -  3723 hrtime pts/0    00:00:00 a.out
1 S  1000  19647  19648   1856  0  80   0 -  3723 hrtime pts/0    00:00:00 a.out
1 S  1000  19649  19649  19647  0  80   0 -  3723 wait   pts/0    00:00:00 a.out
0
in start main()
1
0 S  1000  19647  19647   1856  0  80   0 - 20107 wait   pts/0    00:00:00 a.out
1 S  1000  19647  19648   1856  0  80   0 - 20107 hrtime pts/0    00:00:00 a.out
1 S  1000  19649  19649  19647  0  80   0 -  3723 hrtime pts/0    00:00:00 a.out
isscgy@ubuntu:/mnt/os-2020$ 1
1
1
1
ps
   PID TTY          TIME CMD
  1856 pts/0    00:00:04 bash
 19649 pts/0    00:00:00 a.out
 19656 pts/0    00:00:00 ps
isscgy@ubuntu:/mnt/os-2020$ 1
1
^C
isscgy@ubuntu:/mnt/os-2020$ 1
pk1
ill -f a1
.out
isscgy@ubuntu:/mnt/os-2020$
```

pid=19647, spid=19647: 父进程main()线程
pid=19647, spid=19648: 父进程thread_worker线程
pid=19649, spid=19649: 子进程thread_worker线程

父进程结束，它的线程也结束了；
但子进程仍然运行，输出1，^C也不能终止；
必须使用pkill强制杀死它
$pkill -f a.out

Threading Issues

- **信号处理**
  - UNIX系统中使用信号通知进程某个特定事件已发生。
    - 信号可以同步或异步接收
  - 所有信号应遵循以下模式：
    - 特定事件的发生会产生一个信号。
    - 信号被传送到进程。
    - 信号一旦发出，必须被处理。
  - 信号由这两个信号处理程序之一处理
    - 内核运行的默认处理程序
    - 可以覆盖默认处理程序的用户定义处理程序。
  - 对于单线程，一个信号被传送到一个进程。
  - 多线程的信号应该在哪里传递？
    - 将信号传递到适用该信号的线程。
    - 将信号传递给进程中的每个线程。
    - 将信号传递给进程中的某些线程。
    - 指定一个特定线程来接收进程的所有信号。
    - 稍后讨论CLONE_THREAD克隆线程标志。

Threading Issues

- **信号处理**
  - 实例
    - 用于向指定进程传递信号的标准UNIX函数

```
int kill(pid_t pid, int signal)
```

    - POSIX Pthreads函数pthread_kill()允许将信号传递到指定的线程

```
int pthread_kill(pthread_t ptid, int signal)
```

信号处理

- Linux

```
isscgy@ubuntu:/mnt/hgfs/os-2020$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
isscgy@ubuntu:/mnt/hgfs/os-2020$
```

- 1-31号是常规信号（非实时信号），没有阻塞队列，多次产生时只有一次记录。
- 32-64号是实时信号，支持排队策略。

信号处理

- Linux–数据结构

```
struct sigaction {
    union {
        __sighandler_t _sa_handler;
        void (*_sa_sigaction)(int, struct siginfo *, void *);
    } _u
    sigset_t sa_mask;
    unsigned long sa_flags;
}

struct sigaction {
    void (*sa_handler)(int); /* func name | SIG_IGN | SIG_DFL */
    void (*sa_sigaction)(int, siginfo_t *, void *); /* 很少用 */
    sigset_t sa_mask; /* 64位掩码，仅对操作有效*/
    int sa_flags;     /* 0: 屏蔽自身 */
    void (*sa_restorer)(void); /*弃置*/
};
```

- SIG_IGN和SIG_DFL是内核中定义的句柄，这意味着忽略信号或默认情况下处理。

信号处理

- Linux–信号注册

```
int sigaction(int signum, const struct sigaction *act,
struct sigaction *oldact);
/* signum: 除9 SIGKILL和19 SIGSTOP外的注册信号;
   *act: 新sigaction;
   *oldact: 已保存的旧sigaction, 如不用可设为NULL */
```

- Linux–设置掩码sigset_t sa_mask

```
int sigemptyset(sigset_t *set) /* clear all */
int sigfillset(sigset_t *set)  /* set all */
int sigaddset(sigset_t *set, int signum)   /* add */
int sigdelset(sigset_t *set, int signum)   /* delete */
int sigismember(sigset_t *set, int signum) /* query */
```

- Linux–发送信号

```
int sigqueue(pid_t pid, int signo, const union sigval
value);
int kill(pid_t pid, int signal)
int tgkill(int tgid, int tid, int sigal);
int tkill(int tid, int sigal); /* tgkill()的前身,废弃 */
```


信号处理

- 算法. 13-8-sigaction-demo.c (1)

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>

void my_handler(int signo)  /*用户信号处理程序*/
{
    printf("\nhere is my_handler");
    printf("\nsignal catched: signo = %d", signo);
    printf("\nCtrl+\\ is masked");
    printf("\nsleeping for 10 seconds ... \n");
    sleep(10);
    printf("my_handler finished\n");
    printf("after returned to the main(), Ctrl+\\ is unmasked\n");
    return;
}
```

信号处理

- 算法. 13-8-sigaction-demo.c (2)

```
int main(void)
{
    int ret;
    struct sigaction newact;

    newact.sa_handler = my_handler; /* 设置用户定义的处理程序 */
    sigemptyset(&newact.sa_mask);   /* 清除屏蔽掩码 */
    sigaddset(&newact.sa_mask, SIGQUIT); /* sa_mask, 设置屏蔽signo=3
                                            (SIGQUIT:Ctrl+\) */
    newact.sa_flags = 0; /* 缺省标志 */

    printf("now start catching Ctrl+c\n");

    ret = sigaction(SIGINT, &newact, NULL);/*注册signo=2 (SIGINT:Ctrl+C)*/
    if (ret == -1) {
        perror("sigaction()");
        exit(1);
    }

    while (1);
    return 0;
}
```


信号处理

- 算法. 13-8-sigaction-demo.c (2)

```
int main(void)
{
    int ret;
isscgy@ubuntu:/mnt/os-2020$ gcc alg.13-8-sigaction-demo.c
isscgy@ubuntu:/mnt/os-2020$ ./a.out
now start catching Ctrl+c
^C
here is my_handler
signal catched: signo = 2
Ctrl+\ is masked
sleeping for 10 seconds ...
^\my_handler finished
after returned to the main(), Ctrl+\ is unmasked
Quit (core dumped)
isscgy@ubuntu:/mnt/os-2020$
        exit(1);
    }

    while (1);
    return 0;
}
```

按^C触发my_handler函数执行；
执行中按^\，因signo 2（Ctrl+\）被屏蔽，此信号被延迟执行，直到my_handler处理程序结束后才被处理；
SIGQUIT默认处理是退出并内核转储

线程取消

- 线程取消涉及在线程完成之前终止线程（这里称为目标线程）。
- 通过两种方法取消目标：
  - 异步取消
    - 立即终止目标线程。
  - 延期取消
    - 允许目标线程定期检查是否应取消它。
- 取消的困难发生在以下情况：
  - 资源已分配给已取消的线程，或者
  - 线程在更新与其他线程共享的数据时被取消

线程取消

- 实例
  - 在POSIX Pthreads中，pthread_cancel()函数用于取消指定的线程，目标线程的标识符作为参数传递给函数。
  - 下面的代码演示了如何创建和取消线程。

```
pthread_t ptid;
  /* create the thread */
pthread_create(&ptid, 0, &thread_worker, NULL);
    . . .
  /* cancel the thread */
pthread_cancel(ptid);
```

线程取消

- Pthreads取消模式
  - Pthreads支持三种取消模式。每个模式都定义为一个状态和一个类型。
    - 关闭模式
      - State = PTHREAD_CANCEL_DISABLE;
      - Type没有定义
    - 延迟模式
      - State = PTHREAD_CANCEL_ENABLE;
      - Type = PTHREAD_CANCEL_DEFERRED
    - 异步模式
      - State = PTHREAD_CANCEL_ENABLE;
      - Type = PTHREAD_CANCEL_ASYNCHRONOUS
  - 默认取消类型为延迟取消。取消请求一直挂起，直到线程到达取消点。

线程取消

- Pthreads取消点
  - 取消点
    - POSIX标准中，pthread_join()、pthread_testcancel()、pthread_cond_wait()、pthread_cond_timedwait()、sem_wait()、sigwait()函数以及可能导致系统阻塞的系统调用read()、write()都是取消点。
  - 函数void pthread_testcancel(void)可用于在线程中设置取消点，或取消其取消请求被延迟的线程
  - 例子：Linux中read()/write()内未实现取消点，需显式设置

```
pthread_testcancel(); /*测试取消*/
retcode = read(fd, buffer, length); /*C函数内未实现取消点*/
pthread_testcancel(); /*再次测试取消*/
```

Threading Issues

- **线程取消**
  - **与线程取消有关的Pthreads函数**
    - `int pthread_create(pthread_t *restrict ptid, const pthread_attr_t *restrict_attr, void*(*start_rtn)(void*), void *restrict arg);`
    - `int pthread_cancel(pthread_t ptid);`
    - `void pthread_testcancel(void);`
    - `int pthread_setcancelstate(int state, int *oldstate);`
    - `int pthread_setcanceltype(int type, int *oldtype);`
    - `void pthread_cleanup_push(void (*routine) (void *), void *arg);`
    - `void pthread_cleanup_pop(int execute);`