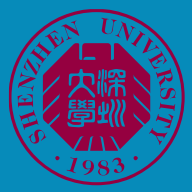



深圳大学 计算机与软件学院

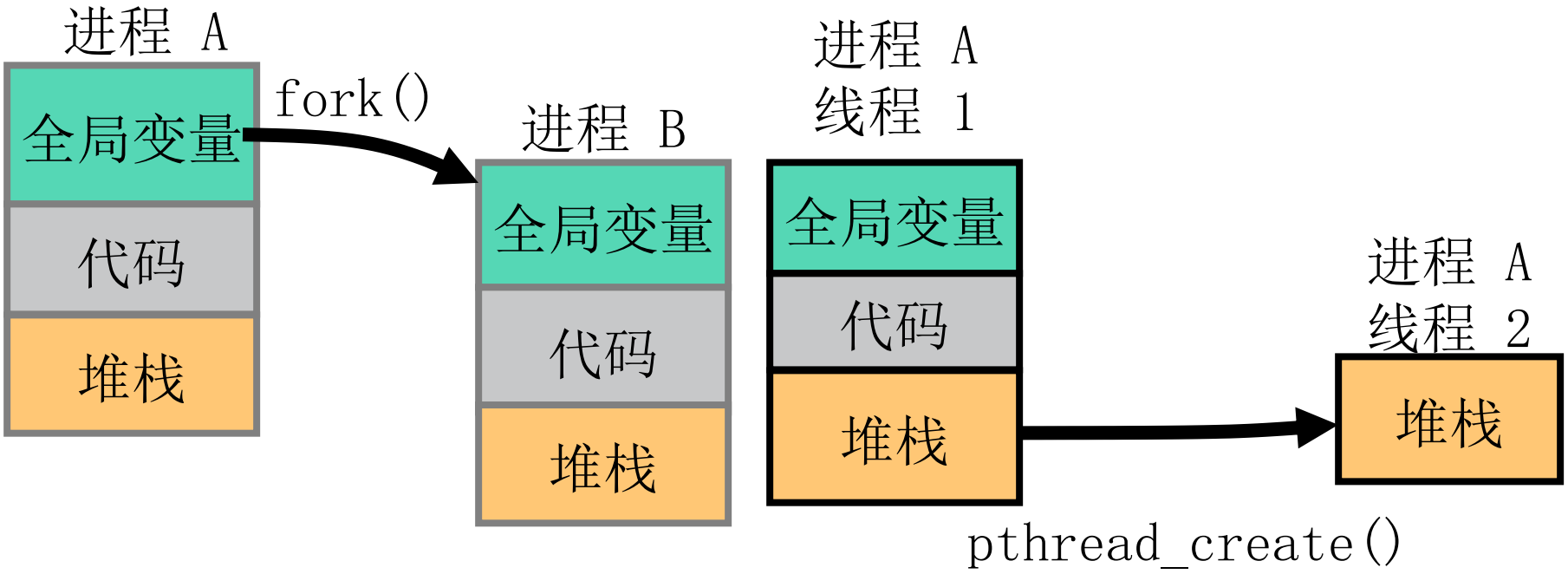
College of Computer Science and Software Engineering of Shenzhen University

系统编程

基于TaiShan服务器/openEuler OS 的实践

第三讲：多线程编程 – 数据安全

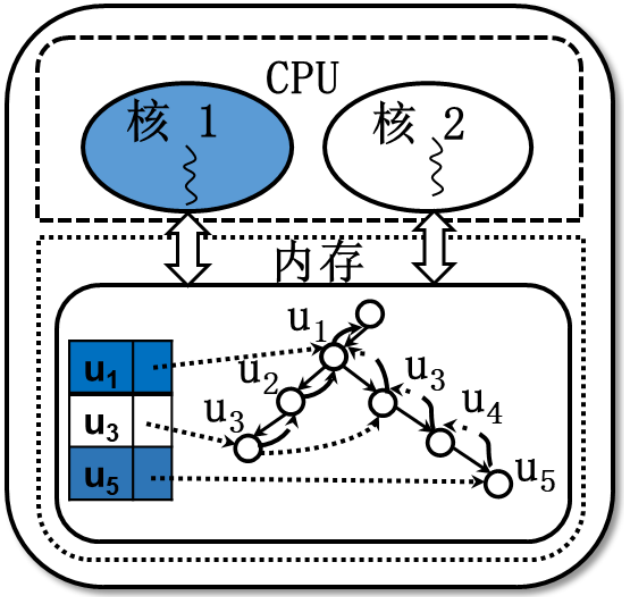
线程模型

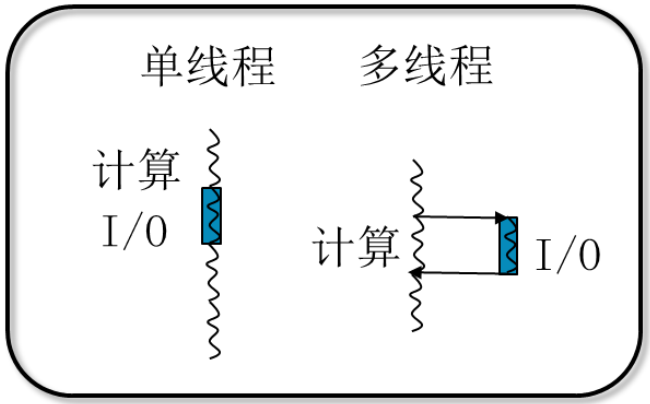


进程创建的代价相对高
(时间 & 内存).

线程的优点 - 提高系统吞吐量

- **分解问题**
  - 问题表示更抽象
  - 问题解决更模块化
    - 高内聚，低耦合
    - 充分利用多核
- **分离I/O和计算**
  - 降低用户响应时间
- **共享存储地址空间和文件描述符**
  - 减少系统资源的占用





线程数据的挑战

- 请问以下代码的运行结果是？

```
#include <stdio.h>
#include <pthread.h>
#define THREAD_NUM 4

void *printArgument(void *argument)
{
        int *p = (int *)argument;
        int printContent = *p;

        printf("Thread number %d\n",printContent);
}

void main()
{
        pthread_t threadId[THREAD_NUM];
        int i;
        for (i = 0; i < THREAD_NUM; i++)
                pthread_create(&threadId[i],NULL,printArgument,&i);

        for (i = 0; i < THREAD_NUM; i++)
                pthread_join(threadId[i],NULL);
}
```

线程数据的挑战

- 运行结果

```
[yuhong@FedoraDVD13 thread]$ ./correctionNeeded
Thread number 0
Thread number 0
Thread number 0
Thread number 0
[yuhong@FedoraDVD13 thread]$
```

- 期望的结果

```
[yuhong@FedoraDVD13 thread]$ ./correctionNeeded1
Thread number 3
Thread number 2
Thread number 1
Thread number 0
[yuhong@FedoraDVD13 thread]$
```

线程数据的挑战

- 如何修改代码？

```
#include <stdio.h>
#include <pthread.h>
#define THREAD_NUM 4

void *printArgument(void *argument)
{
        int *p = (int *)argument;
        int printContent = *p;

        printf("Thread number %d\n",printContent);
}

void main()
{
        pthread_t threadId[THREAD_NUM];
        int i;
        for (i = 0; i < THREAD_NUM; i++)
                pthread_create(&threadId[i],NULL,printArgument,&i);

        for (i = 0; i < THREAD_NUM; i++)
                pthread_join(threadId[i],NULL);
}
```

- 原因
  - 同一进程的所有线程共享进程的数据空间
  - 全局变量为所有线程共享
- 数据竞争 & 结果不确定

线程数据的挑战 – 解决方案

- 方案一

```
#include <stdio.h>
#include <pthread.h>
#define THREAD_NUM 4

void *printArgument(void *argument)
{
        int *p = (int *)argument;
        int printContent = *p;

        printf("Thread number %d\n",printContent);
}

void main()
{
        pthread_t threadId[THREAD_NUM];
        int i;
        for (i = 0; i < THREAD_NUM; i++)
                pthread_create(&threadId[i],NULL,printArgument,&i);

        for (i = 0; i < THREAD_NUM; i++)
                pthread_join(threadId[i],NULL);
}
```

int array[3];

array[i]=i;

&array[i];

方案二 – 线程私有数据

- **线程私有数据(Thread-specific Data, TSD)特点**
  - 不是局部变量
    - 线程函数及其调用到的函数都可访问
  - 不是全局变量
    - 仅线程函数及其调用到的函数可访问
    - 其他线程函数不可访问

线程私有数据模型

void *TSD[keys][tids]

keys	tid_1	tid_2	…	tid_k
K_1				
…				
K_m				

- 一键多值数据模型
  - 公用健
    - 数据访问通过健值
  - 不同线程通过同一键值映射到不同的数据
  - 数据类型是指向void的指针，可以指向任意类型数据
- errno 被重定义为线程私有数据

线程私有数据操作 - 键的创建 & 删除

- 创建

```
int pthread_key_create(pthread_key_t *key,
                       void (*destructor) (void *));
```

key：创建一个线程私有数据键，其实质是将TSD结构数组中的某一项设置为“in_use”，并将其索引赋给*key，也成为从TSD池中分配一项

destructor：析构函数，释放键key对应的线程私有内存块,线程结束时系统自动调用

- 成功： 返回0
- 出错： 返回大于0的出错代码

```
static struct
   pthread_key_struct_pthread_keys[PTHREAD_KEYS_MAX] =
   {{0,NULL}};
```

- 删除

```
int pthread_key_delete(pthread_key_t key);
```

线程私有数据操作 - 键值的读取 & 设置

- 从一个线程私有数据键读取其值

```
#include <pthread.h>
void *pthread_getspecific(pthread_key_t key);
```

- 为一个线程私有数据键设置值

```
#include <pthread.h>
int pthread_setspecific(pthread_key_t key, const
    void *value);
```

void *TSD[keys][tids]

keys	tid_1	tid_2	…	tid_k
K_1	0	5		
…				
K_m				

yuhong@FedoraDVD13:~/C程序/thread/exitStatus

文件(F) 编辑(E) 查看(V) 终端(T) 帮助(H)

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

pthread_key_t key;

void *thread2(void *arg)
{
        int tsd = 5;
        printf("thread %u is running\n",pthread_self());
        pthread_setspecific(key,(void *)tsd);
        printf("thread %u returns %d\n",pthread_self(),pthread_getspecific(key));
}

void *thread1(void *arg)
{
        int tsd = 0;
        pthread_t threadid2;

        printf("thread %u is running\n",pthread_self());
        pthread_setspecific(key,(void *)tsd);
        pthread_create(&threadid2,NULL,thread2,NULL);
        pthread_join(threadid2,NULL);
        printf("thread %u returns %d\n",pthread_self(),pthread_getspecific(key));
}

int main(void)
{
        pthread_t threadid1;
        printf("main thread begins running\n");
        pthread_key_create(&key,NULL);
        pthread_create(&threadid1,NULL,thread1,NULL);
        pthread_join(threadid1,NULL);
        pthread_key_delete(key);
        printf("main thread exit\n");
        return 0;
}
```

void *TSD[keys][tids]

keys	tid_1	tid_2	…	tid_k
K_1	0	5		
…				
K_m				

```
[yuhong@FedoraDVD13 exitStatus]$ gcc -lpthread -o tsd tsd.c
[yuhong@FedoraDVD13 exitStatus]$ ./tsd
main thread begins running
thread 3078540144 is running
thread 3068050288 is running
thread 3068050288 returns 5
thread 3078540144 returns 0
main thread exit
```

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

pthread_key_t key1;
pthread_key_t key2;

struct specific_data{
    int month;
    float salary;
}monthly_data;

float bonus;

void setspecificincome(){
    pthread_setspecific(key1,&monthly_data);
    pthread_setspecific(key2,&bonus);
}

int changedata(int mon, float sal, float bonusvar)
{
    bonus = bonusvar;
    monthly_data.month = mon;
    monthly_data.salary = sal;

    return 0;
}

int printdata()
{

    struct specific_data *sd;
    float *tempbonus;

    sd = (struct specific_data *)pthread_getspecific(key1);
    tempbonus = (float *)pthread_getspecific(key2);
    printf("Thread %u month: %d salary: %f bonus: %f\n", pthread_self(),sd->month,sd->salary,*tempbonus);
}

void *manager_monsal(void *arg)
{
    printf("Thread %u is running ... \n",pthread_self());
    changedata(2,10000.00, 800.00);
    setspecificincome();
    printdata();
}

void *member_monsal(void *arg)
{
    printf("Thread %u is runnning ... \n",pthread_self());
    changedata(3,5000.00,400.00);
    setspecificincome();
    printdata();
}
```

```c
int create_key(){
    pthread_key_create(&key1,NULL);
    pthread_key_create(&key2,NULL);
    return 0;
}

int delete_key(){
    pthread_key_delete(key1);
    pthread_key_delete(key2);
    return 0;
}

int main(void)
{
    pthread_t tid1,tid2;

    create_key();
    pthread_create(&tid1,NULL,manager_monsal,NULL);
    pthread_create(&tid2,NULL,member_monsal,NULL);
    pthread_join(tid1,NULL);
    pthread_join(tid2,NULL);
    delete_key();
    return 0;
}
```

```
[szu@taishan02-vm-10 threads]$ gcc -lpthread -o privatedata privatedata.c
[szu@taishan02-vm-10 threads]$ ./privatedata
Thread 156955104 is running ...
Thread 156955104 month: 2 salary: 10000.000000 bonus: 800.000000
Thread 148500960 is runnning ...
Thread 148500960 month: 3 salary: 5000.000000 bonus: 400.000000
```

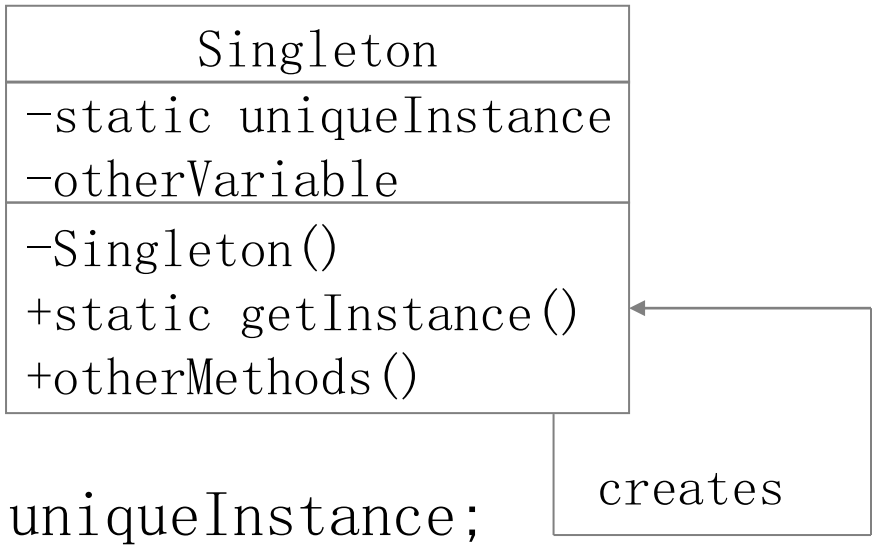
一次性初始化

- 应用场景
  - 线程一个键应只被创建一次
  - 服务/系统初始化工作
  - Java的单例模式
- 如何实现一次性初始化?

```
int flag = 0;
int resource;
void init_once(){
        if (flag == 0){
                flag = 1;
                ……   //初始化工作
        }
}
```

人生若只如初见☺

Java单例模式




```
public class Singleton{
   private static Singleton uniqueInstance;
   private Singleton(){};

   public static synchronized Singleton getInstance()
   {
       if (uniqueInstance==null) {
            uniqueInstance=new Singleton();
       }
       return uniqueInstance;
   }
}
```

线程数据的挑战 （二）

yuhong@FedoraDVD13:~/C程序

文件(F) 编辑(E) 查看(V) 终端(T) 帮助(H)

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

int x = 1;

void thread(void)
{
        x = x + 1;
        printf("x is %d\n",x);
}

int main(void)
{
        pthread_t id;
        int i,ret;
        ret=pthread_create(&id,NULL,(void *) thread,NULL);
        if (ret != 0){
                printf("Create pthread error!\n");
                exit(1);
        }

        thread();

        pthread_join(id,NULL);
        return 0;
}
```

yuhong@FedoraDVD13

文件(F) 编辑(E) 查看(V) 终端(T) 帮助(H)

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

int x = 1;

void thread(void)
{
        x = x + 1;
        printf("x is %d\n",x);
}

int main(void)
{
        pid_t id;
        int i,ret;
        ret=fork();
        if (ret < 0){
                printf("Create pthread error!\n");
                exit(1);
        }

        thread();
        return 0;
}
```

线程数据的挑战 （二）

```
int x = 1;

void* func(void* p){
    x = x + 1;
    printf("x is %d\n", x);
    return NULL;
}
```

```
void* func(void* p){
    x = x + 1;
    printf("x is %d\n", x);
    return NULL;
}
```

time

可能的运行结果1：

Output:

```
x is 2
x is 3
```

12

线程数据的挑战 （二）

```
int x = 1;

void* func(void* p){
    x = x + 1;
    printf("x is %d\n", _);
    return NULL;
}
```

x

```
void* func(void* p){
    x = x + 1;
    printf("x is %d\n", x);
    return NULL;
}
```

time

Output:

x is 3

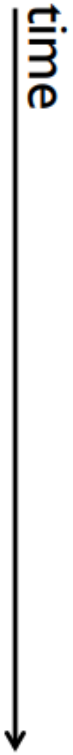
x is 2

可能的运行结果2：

线程数据的挑战 （二）

```
int x = 1;

void* func(void* p){
    x = x + 1;

    printf("x is %d\n", x);
    return NULL;
}
```

```
void* func(void* p){

    x = x + 1;

    printf("x is %d\n", x);
    return NULL;
}
```



可能的运行结果3：

Output:

```
x is 3
x is 3
```

线程数据的挑战 （二）

```
int x = 1;

void* func(void* p){
        x + 1
        ↓
    x = _____;
    printf("x is %d\n", x);
    return NULL;
}
```

```
void* func(void* p){
        x + 1
        ↓
    x = _____;
    printf("x is %d\n", x);
    return NULL;
}
```

可能的运行结果4：

Output:

```
x is 2
x is 2
```

线程数据的挑战（二）- 一次性初始化 - 解决方法

- 多线程环境下如何实现一次性初始化?
  - 互斥量
  - pthread_once

```
pthread_once_t once_control=PTHREAD_ONCE_INIT;
int pthread_once(pthread_once_t *once_control,
                 void(*init_routine)(void));
once_control          控制变量
init_routine          初始化函数
成功：返回0
失败：返回大于0的错误编号
```

```
[szu@taishan02-vm-10 threads]$ cat initonce.c
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

pthread_once_t once = PTHREAD_ONCE_INIT;

void run_once(void)
{
        printf("Function run_once() is executed by thread %u\n",pthread_self());
}

void *act(void *arg)
{
        printf("Thread %u is running...\n", pthread_self());
        pthread_once(&once, run_once);
        printf("Thread %u ends.\n",pthread_self());
}

int main()
{
        pthread_t threadId1, threadId2;

        if (pthread_create(&threadId1, NULL, act, NULL) < 0) {
                perror("Fail to create thread 1...\n");
                exit(-1);
        }
        if (pthread_create(&threadId2, NULL, act, NULL) < 0){
                perror("Fail to creatge thread 2...\n");
                exit(-1);
        }
        pthread_join(threadId1,NULL);
        pthread_join(threadId2,NULL);
}
```

一次性初始化应用例子（一）

```
[szu@taishan02-vm-10 threads]$ gcc -lpthread -o initonce initonce.c
[szu@taishan02-vm-10 threads]$ ./initonce
Thread 2164519392 is running...
Function run_once() is executed by thread 2164519392
Thread 2156065248 is running...
Thread 2156065248 ends.
Thread 2164519392 ends.
```

一次性初始化应用例子（二）

```c
/*
 * tsd_once.c
 *
 * 演示在多线程编程中使用pthread_once来实行一次性初始化
 */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

/*
 * 使用结构数据作为线程私有数据键
 */
typedef struct tsd_tag{
        pthread_t thread_id;
        char *string;
}tsd_t;

pthread_key_t tsd_key;
pthread_once_t key_once =PTHREAD_ONCE_INIT;

/*
 * 一次性初始化子程序，通过pthread_once控制实现
 */
void once_routine(void)
{
        int status;

        printf("initializing key\n");
        status = pthread_key_create(&tsd_key,NULL);
        if (status != 0) printf("Fail to create a key...\n");
}
```

一次性初始化应用例子（二）

```c
/*
 * 线程启动使用pthread_once的子程序来动态地创建一个线程私有数据键
 */

void *thread_routine(void *arg)
{
        tsd_t *value;
        int status;

        status = pthread_once(&key_once,once_routine);//一次性初始化
        if (status !=0 ) printf("Fail to init once...\n");
        value = (tsd_t *)malloc(sizeof(tsd_t));
        if (value == NULL) printf("Fail to allocate key value...\n");
        status = pthread_setspecific(tsd_key,value);
        if (status != 0) printf("Fail to set tsd...\n");
        printf("%s set tsd value %p\n",arg,value);
        value->thread_id = pthread_self();
        value->string = (char *)arg;
        value = (tsd_t*)pthread_getspecific(tsd_key);
        printf("%s starting...\n",value->string);
        sleep(2);
        value = (tsd_t*)pthread_getspecific(tsd_key);
        printf("%s done...\n",value->string);
        return NULL;
}

void main(int argc, char *argv[])
{
        pthread_t thread1,thread2;
        int status;

        status = pthread_create(&thread1,NULL,thread_routine,"thread 1");
        if (status != 0) printf("Fail to create thread1...");
        status = pthread_create(&thread2,NULL,thread_routine,"thread 2");
        if (status != 0) printf("Fail to create thread2...\n");
        pthread_join(thread1,NULL);
        pthread_join(thread2,NULL);
}
```

一次性初始化应用例子（二）

```
[yuhong@FedoraDVD13 exitStatus]$ gcc -pthread -o tsd_once tsd_once.c
[yuhong@FedoraDVD13 exitStatus]$ ./tsd_once
initializing key
thread 2 set tsd value 0xb6300468
thread 2 starting...
thread 1 set tsd value 0xb6100468
thread 1 starting...
thread 2 done...
thread 1 done...
[yuhong@FedoraDVD13 exitStatus]$
```

线程数据的挑战(二)-运行期多线程多次共同读写数据

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

#define NLOOP 5000

int counter;/*incremented by threads*/

void *increase(void *vptr);

int main(int argc, char **argv)
{
        pthread_t threadIdA,threadIdB;

        pthread_create(&threadIdA,NULL,&increase,NULL);
        pthread_create(&threadIdB,NULL,&increase,NULL);

        /*wait for both threads to terminate*/
        pthread_join(threadIdA,NULL);
        pthread_join(threadIdB,NULL);

        return 0;
}

void *increase(void *vptr)
{
        int i, val;

        for (i = 0; i < NLOOP; i++){
                val = counter;
                printf("%x: %d\n", (unsigned int)pthread_self(),val+1);
                counter = val +1;
        }

        return NULL;
}
```

多次运行结果不一致

```
b7713b70: 5075
b7713b70: 5076
b7713b70: 5077
b7713b70: 5078
b7713b70: 5079
b7713b70: 5080
b7713b70: 5081
b7713b70: 5082
b7713b70: 5083
b7713b70: 5084
b7713b70: 5085
b7713b70: 5086
b7713b70: 5087
b7713b70: 5088
b7713b70: 5089
b7713b70: 5090
b7713b70: 5091
b7713b70: 5092
b7713b70: 5093
b7713b70: 5094
b7713b70: 5095
b7713b70: 5096
b7713b70: 5097
b7713b70: 5098
b7713b70: 5099
b7713b70: 5100
b7713b70: 5101
b7713b70: 5102
b7713b70: 5103
b7713b70: 5104
b7713b70: 5105
b7713b70: 5106
b7713b70: 5107
b7713b70: 5108
b7713b70: 5109
b7713b70: 5110
b7713b70: 5111
b7713b70: 5112
b7713b70: 5113
b7713b70: 5114
b7713b70: 5115
b7713b70: 5116
b7713b70: 5117
b7713b70: 5118
b7713b70: 5119
[yuhong@FedoraDVD13 Second]$
```

第一次运行

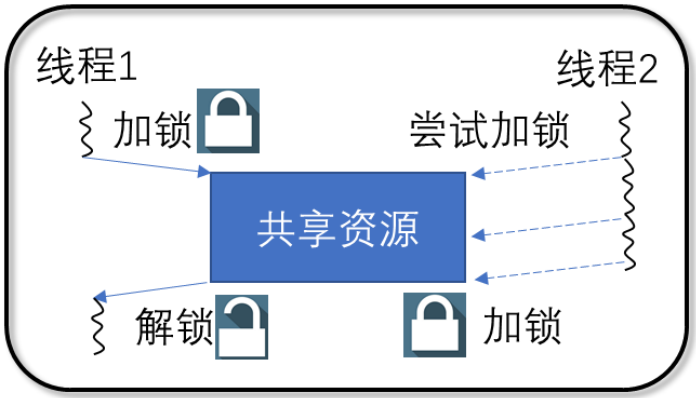
```
b774fb70: 5332
b774fb70: 5333
b774fb70: 5334
b774fb70: 5335
b774fb70: 5336
b774fb70: 5337
b774fb70: 5338
b774fb70: 5339
b774fb70: 5340
b774fb70: 5341
b774fb70: 5342
b774fb70: 5343
b774fb70: 5344
b774fb70: 5345
b774fb70: 5346
b774fb70: 5347
b774fb70: 5348
b774fb70: 5349
b774fb70: 5350
b774fb70: 5351
b774fb70: 5352
b774fb70: 5353
b774fb70: 5354
b774fb70: 5355
b774fb70: 5356
b774fb70: 5357
b774fb70: 5358
b774fb70: 5359
b774fb70: 5360
b774fb70: 5361
b774fb70: 5362
b774fb70: 5363
b774fb70: 5364
b774fb70: 5365
b774fb70: 5366
b774fb70: 5367
b774fb70: 5368
b774fb70: 5369
b774fb70: 5370
b774fb70: 5371
b774fb70: 5372
b774fb70: 5373
b774fb70: 5374
b774fb70: 5375
b774fb70: 5376
[yuhong@FedoraDVD13 Second]$
```

第二次运行

运行期多线程多次读写共享数据 – 同步 & 互斥锁

- **线程同步**
  - 协调多个相关线程的执行次序
  - 并发线程间有效共享资源和相互协作
  - 运行结果具有可再现性

- **实现方式之一：互斥锁/互斥量**
  - 读写数据前先尝试加锁
  - 加锁成功后读写数据
  - 读写完毕后解锁
  - ✓同一时间只有一个线程持有该锁，即只允许一个线程读写共享数据



互斥锁pthread_mutex_t mutex的初始化

- 静态初始化：全局或加static修饰的变量，使用宏

  mutex = PTHREAD_MUTEX_INITIALIZER;

- 动态初始化：局部变量，使用pthread_mutex_init()函数

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t *restrict attr)
```

- 返回值
  - 成功：返回0，且mutex的状态为unlocked
  - 失败：返回非0出错代码

pthread_mutex_init()的参数

- mutex：结构体类型，简化看作整数
  - mutex可取两值：0或1
  - restrict关键字限制修改该指针指向内存内容的操作只能通过该指针完成

- attr：互斥锁属性,包括作用域和类型
  - 作用域
    - ◆PTHREAD_PROCESS_SHARED ：系统范围内多进程的线程间共享
      - ➢在共享内存中创建互斥锁和设置互斥锁属性
    - ◆PTHREAD_PROCESS_PRIVATE：进程专用锁
      - ➢同一个进程的线程间共享

```
pthread_mutexattr_setpshared(&mattr,PTHREAD_PROCESS_PRIVATE);
pthread_mutexattr_getpshared(&mattr, &pshared);
```

pthread_mutex_init()的参数

- attr：互斥锁属性,包括作用域和类型
  - 类型
    - PTHREAD_MUTEX_NORMAL：不检测死锁
      - 未解锁，再次锁定该互斥锁：死锁
    - PTHREAD_MUTEX_ERRORCHECK：检测死锁
      - 未解锁，再次锁定该互斥锁：返回错误
    - PTHREAD_MUTEX_RECURSIVE：未解锁，可再次锁定该锁
      - n次锁定互斥锁,n次解锁才可释放该锁
    - PTHREAD_MUTEX_DEFAULT
      - 不确定行为：以递归方式锁定此类型互斥锁、解除非调用线程锁定的此类型互斥锁、解除尚未锁定的此类型互斥锁
      - 允许在实现中将该互斥锁映射到其他类型之一
        - Solaris 线程，映射到 PTHREAD_PROCESS_NORMAL

```
int pthread_mutexattr_settype(pthread_mutexattr_t *attr ,int type);
ret = pthread_mutexattr_getpshared(&mattr, &pshared);
```

互斥锁操作函数

#include <pthread.h>

- 加锁
  - 锁的状态为unlocked，当前线程获得锁，将其改为locked，并进入临界区
  - 锁的状态为locked,当前线程阻塞

  int pthread_mutex_lock(pthread_mutex_t *mutex);
- 测试加锁:加琐,如果失败不阻塞

  int pthread_mutex_trylock(pthread_mutex_t *mutex);
- 解锁：同时将阻塞在该锁上的所有线程全部唤醒

  int pthread_mutex_unlock(pthread_mutex_t *mutex);
- 释放锁:锁的状态为unlocked,成功返回（Linux）
  - 锁占用资源的系统，释放对相应资源

  int pthread_mutex_destroy(pthread_mutex_t *mutex);

一个不加锁的程序

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>

void *thread_function(void *arg);

int run_now = 1;        /*用 run_now 代表共享资源*/
int main()
{
        int print_count1 = 0;
        pthread_t a_thread;

        if (pthread_create(&a_thread,NULL,thread_function,NULL)!=0){
                perror("Thread creation failed");
                exit(1);
        }

        while (print_count1++<5){
                if (run_now == 1){
                        printf("main thread is running\n");
                        run_now = 2;
                } else {
                        printf("main thread is sleep\n");
                        sleep(1);
                }
        }
        pthread_join(a_thread,NULL);
        exit(0);
}

void *thread_function(void *arg){
        int print_count2 = 0;

        while (print_count2++<5){
                if (run_now == 2){
                        printf("function thread is running\n");
                        run_now = 1;
                } else {
                        printf("function thread is sleep\n");
                        sleep(1);
                }
        }
        pthread_exit(NULL);
}
```

如果run_now为1，就把它修改为2

如果run_now为2，就把它修改为1

```
[yuhong@FedoraDVD13 Second]$ ./noMutex
main thread is running
main thread is sleep
function thread is running
function thread is sleep
main thread is running
main thread is sleep
function thread is running
function thread is sleep
main thread is running
function thread is running
```

一个不加锁的程序

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>

void *thread_function(void *arg);

int run_now = 1;          /*用 run_now 代表共享资源*/
int main()
{
        int print_count1 = 0;
        pthread_t a_thread;

        if (pthread_create(&a_thread,NULL,thread_function,NULL)!=0){
                perror("Thread creation failed");
                exit(1);
        }

        while (print_count1++<5){
                if (run_now == 1){
                        printf("main thread is running\n");
                        run_now = 2;
                } else {
                        printf("main thread is sleep\n");
                        sleep(1);
                }
        }
        pthread_join(a_thread,NULL);
        exit(0);
}

void *thread_function(void *arg){
        int print_count2 = 0;

        while (print_count2++<5){
                if (run_now == 2){
                        printf("function thread is running\n");
                        run_now = 1;
                } else {
                        printf("function thread is sleep\n");
                        sleep(1);
                }
        }
        pthread_exit(NULL);
}
```

repeat

entry section

critical section;

exit section

remainder section;

until false;

同步机制的规则

- 空闲让进
- 忙则等待
- 有限等待
- 让权等待

加锁后的程序

```c
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <pthread.h>

void *thread_function(void *arg);
int run_now = 1;          /*用run_now代表共享资源*/
pthread_mutex_t work_mutex; /*定义互斥量*/

int main()
{
        int res;
        int print_count1 = 0;
        pthread_t a_thread;

        if (pthread_mutex_init(&work_mutex,NULL)!=0){
                perror("Mutex init failed");exit(1);
        }

        if (pthread_create(&a_thread,NULL,thread_function,NULL)!=0){
                perror("Thread creation failed");exit(1);
        }

        if (pthread_mutex_lock(&work_mutex)!=0){
                perror("Lock failed");exit(1);
        } else  printf("main lock\n");

        while (print_count1++<5){
                if (run_now == 1){
                        printf("main thread is running\n");
                        run_now = 2;
                } else {
                        printf("main thread is sleep\n");
                        sleep(1);
                }
        }

        if (pthread_mutex_unlock(&work_mutex)!=0){
                perror("unlock failed");exit(1);
        } else {printf("main unlock\n");}

        pthread_join(a_thread,NULL);
        pthread_mutex_destroy(&work_mutex);
        exit(0);
}
```

练习一

- 请根据例程序继续为线程函数加锁，运行结果如下:

```
[yuhong@FedoraDVD13 Second]$ ./useMutex
main lock
main thread is running
main thread is sleep
main thread is sleep
main thread is sleep
main thread is sleep
main unlock
Function lock
function thread is running
function thread is sleep
```

练习二

- 请设计程序,程序有一个共享变量**counter**，请创建两个线程并发执行**counter++**操作**5000**次。

互斥锁的缺点-加锁和解锁延长运行时间

- 频繁加/解同一互斥锁
  - 延长运行时间
- 没有获得锁的线程阻塞
  - 延长运行时间
- 尽量少用，够用即可
  - 临界区域尽量大
    - 减少加/解锁的频次
  - 包含非临界资源访问的临界区域
    - 延长阻塞线程的等待时间
- 粒度合适的互斥锁

保持“简单”
从来不“简单”☺

Semaphores (信号量）

```
#include <semaphore.h>

int sem_init(sem_t *sem, 0, unsigned int val);} /* s = val */

int sem_wait(sem_t *s);
int sem_post(sem_t *s);
```

- 信号量表示可用资源数
  - sem_wait：申请资源
    - 信号量值>0：信号量值减1，即可用资源数减1
    - 信号量值=0：阻塞
  - sem_post：释放资源
    - 信号量值加1，即可用资源数加1
    - 唤醒一个等待者（如果有）

Semaphores（信号量）

```
sem_t cnt_mutex;

int main(void)
{
    ...
    /* Initialize mutex */
    result = sem_init(&cnt_mutex, 0, 1);
    if (result < 0)
        exit(-1);

    ...

    /* Clean up the semaphore that we're done with */
    result = sem_destroy(&cnt_mutex);
    assert(result == 0);
}
```

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>
sem_t bin_sem;
void *thread_function1(void *arg)
{
    printf("thread_function1--sem_wait\n");
    sem_wait(&bin_sem);
    printf("sem_wait\n");
    while (1){}
}
void *thread_function2(void *arg)
{
    printf("thread_function2-------sem_post\n");
    sem_post(&bin_sem);
    printf("sem_post\n");
    while (1){}
}
```

信号量应用

sem.c

信号量应用

sem.c

```
int main()
{
    int res;
    pthread_t a_thread;
    void *thread_result;

    if ( sem_init(&bin_sem, 0, 0)!= 0 ){
         perror("Semaphore initialization failed…\n"); exit(-1);
    }
    printf("sem_init\n");

    if ( pthread_create(&a_thread, NULL, thread_function1, NULL) != 0 ){
        perror("Thread creation failure…\n"); exit(-1);
    }
    printf("thread_function1\n");

    sleep (5);
    printf("sleep\n");

    if ( pthread_create(&a_thread, NULL, thread_function2, NULL) != 0 ){
        perror("Thread creation failure…\n"); exit(-1);
    }

    while (1) {}
}
```

信号量应用sem.c的可执行文件sem的运行

```
[skywalker@localhost sem]$ ./sem
sem_init
thread_function1
thread_function1--------------sem_wait
sleep
thread_function2--------------sem_post
sem_wait
sem_post
```

条件变量

- 利用线程间共享的全局变量状态变化进行同步
- 条件检测在互斥锁的保护下进行
  - 互斥锁：　短期等待
  - 条件变量：长期等待
- 条件变量使用流程
  - 线程A等待“条件变量的条件成立”而阻塞，并释放等待状态改变的互斥锁
  - 线程B使“条件成立”改变了条件，发信号给相关的条件变量,唤醒线程A及其它等待它的线程
  - 线程A重新获得互斥锁，重新评价条件

条件变量操作函数

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

- **初始化条件变量**
  - **静态分配的条件变量，属性为NULL**

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER
```

  - **动态分配的条件变量**

```
int pthread_cond_init(pthread_cond_t *restrict cond,
                      const pthread_condattr_t *restrict attr)
```

- **清除条件变量并释放为其分配的资源**

```
int pthread_cond_destroy(pthread_cond_t *cond)
```

- **等待条件变量**

```
int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex)
```

条件变量操作函数

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

- 按给定的时间等待条件变量

```
int pthread_cond_timedwait(pthread_cond_t *restrict cond,
                pthread_mutex_t *restrict mutex,
                const struct timespec *restrict abstime)
```

- 唤醒一个等待线程

```
int pthread_cond_signal(pthread_cond_t *cond)
```

- 唤醒阻塞在指定条件变量上的所有线程

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define MAX 5

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

typedef struct {
        char buffer[MAX];
        int how_many;
}BUFFER;

BUFFER share={"",0};
char ch = 'A';

void *readFunc(void *);
void *writeFunc(void *);

int main(void)
{
        pthread_t readThread;
        pthread_t writeThread;

        pthread_create(&readThread,NULL,readFunc,(void *)NULL);
        pthread_create(&writeThread,NULL,writeFunc,(void *)NULL);

        pthread_join(writeThread,(void **)NULL);
        pthread_mutex_destroy(&mutex);
        pthread_cond_destroy(&cond);
        exit(0);
}
```

条件变量典型应用-生产者-消费者

条件变量典型应用-生产者-消费者

```
void *readFunc(void *junk)
{
        int n = 0;
        printf("Read Thread %2d:starting\n",pthread_self());

        while(ch!='Z') {
                pthread_mutex_lock(&mutex);
                if (share.how_many !=MAX){
                        share.buffer[share.how_many++]=ch++;
                        printf("Read Thread %2d: Got char[%c]\n",pthread_self(),ch-1);
                        if (share.how_many == MAX){
                                printf("Read Thread %2d: signaling full\n",pthread_self());
                                pthread_cond_signal(&cond);
                        }
                }
                pthread_mutex_unlock(&mutex);
        }
        sleep(1);
        printf("Read Thread %2d: Exiting\n",pthread_self());
        return NULL;
}

void *writeFunc(void *junk)
{
        int i;
        int n = 0;
        printf("Write Thread %2d: starting\n",pthread_self());

        while(ch!='Z'){
                pthread_mutex_lock(&mutex);
                printf("\nWrite Thread %2d: Waiting\n",pthread_self());
                while(share.how_many != MAX)
                        pthread_cond_wait(&cond,&mutex);
                printf("Write Thread %2d: writing buffer\n",pthread_self());
                for (i = 0; share.buffer[i]&&share.how_many;++i,share.how_many--)
                        putchar(share.buffer[i]);
                pthread_mutex_unlock(&mutex);
        }
        printf("Write Thread %2d: exiting\n",pthread_self());
        return NULL;
}
```

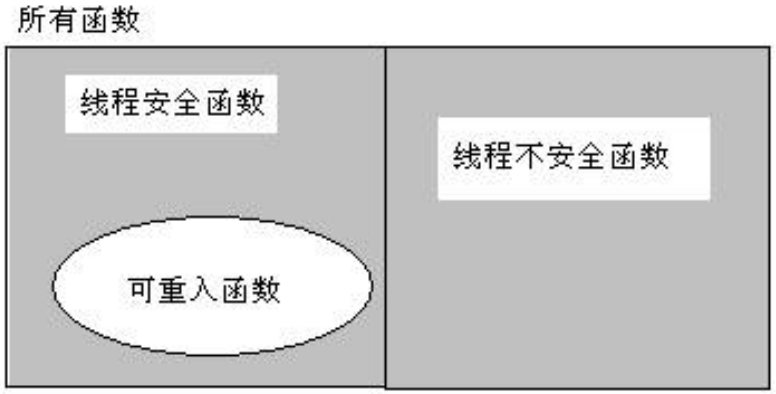
```
Write Thread -1227523216: starting

Write Thread -1227523216: Waiting
Read Thread -1217033360:starting
Read Thread -1217033360: Got char[A]
Read Thread -1217033360: Got char[B]
Read Thread -1217033360: Got char[C]
Read Thread -1217033360: Got char[D]
Read Thread -1217033360: Got char[E]
Read Thread -1217033360: signaling full
Write Thread -1227523216: writing buffer
ABCDE
Write Thread -1227523216: Waiting
Read Thread -1217033360: Got char[F]
Read Thread -1217033360: Got char[G]
Read Thread -1217033360: Got char[H]
Read Thread -1217033360: Got char[I]
Read Thread -1217033360: Got char[J]
Read Thread -1217033360: signaling full
Write Thread -1227523216: writing buffer
FGHIJ
Write Thread -1227523216: Waiting
Read Thread -1217033360: Got char[K]
Read Thread -1217033360: Got char[L]
Read Thread -1217033360: Got char[M]
Read Thread -1217033360: Got char[N]
Read Thread -1217033360: Got char[O]
Read Thread -1217033360: signaling full
Write Thread -1227523216: writing buffer
KLMNO
Write Thread -1227523216: Waiting
Read Thread -1217033360: Got char[P]
Read Thread -1217033360: Got char[Q]
Read Thread -1217033360: Got char[R]
Read Thread -1217033360: Got char[S]
Read Thread -1217033360: Got char[T]
Read Thread -1217033360: signaling full
Write Thread -1227523216: writing buffer
PQRST
Write Thread -1227523216: Waiting
Read Thread -1217033360: Got char[U]
Read Thread -1217033360: Got char[V]
Read Thread -1217033360: Got char[W]
Read Thread -1217033360: Got char[X]
Read Thread -1217033360: Got char[Y]
--More--
```

```
Read Thread -1217033360:starting
Read Thread -1217033360: Got char[A]
Read Thread -1217033360: Got char[B]
Read Thread -1217033360: Got char[C]
Read Thread -1217033360: Got char[D]
Read Thread -1217033360: Got char[E]
Read Thread -1217033360: signaling full
Write Thread -1227523216: writing buffer
ABCDE
Write Thread -1227523216: Waiting
Read Thread -1217033360: Got char[F]
Read Thread -1217033360: Got char[G]
Read Thread -1217033360: Got char[H]
Read Thread -1217033360: Got char[I]
Read Thread -1217033360: Got char[J]
Read Thread -1217033360: signaling full
Write Thread -1227523216: writing buffer
FGHIJ
Write Thread -1227523216: Waiting
Read Thread -1217033360: Got char[K]
Read Thread -1217033360: Got char[L]
Read Thread -1217033360: Got char[M]
Read Thread -1217033360: Got char[N]
Read Thread -1217033360: Got char[O]
Read Thread -1217033360: signaling full
Write Thread -1227523216: writing buffer
KLMNO
Write Thread -1227523216: Waiting
Read Thread -1217033360: Got char[P]
Read Thread -1217033360: Got char[Q]
Read Thread -1217033360: Got char[R]
Read Thread -1217033360: Got char[S]
Read Thread -1217033360: Got char[T]
Read Thread -1217033360: signaling full
Write Thread -1227523216: writing buffer
PQRST
Write Thread -1227523216: Waiting
Read Thread -1217033360: Got char[U]
Read Thread -1217033360: Got char[V]
Read Thread -1217033360: Got char[W]
Read Thread -1217033360: Got char[X]
Read Thread -1217033360: Got char[Y]
Read Thread -1217033360: signaling full
Write Thread -1227523216: writing buffer
UVWXYWrite Thread -1227523216: exiting
[yuhong@FedoraDVD13 Second]$
```

条件变量典型应用-生产者-消费者

线程安全与可重入函数

- **线程安全 (thread-safe)**
  - 一个函数是线程安全的当且仅当其被多个并发进程反复调用时，结果一直正确
- **线程不安全 (thread-unsafe)**
  - 当一个函数不是线程安全的,则它就是线程不安全的
- **可重入函数**
  - 线程安全函数的一种
  - 不引用任何多线程共享数据



- **可重入函数 vs. 不可重入的线程安全函数**
  - 不需要同步操作，效率高

四类不安全的线程函数

- 不保护共享变量的函数
- 保持跨越多个调用的状态函数
- 返回指向静态变量指针的函数
- 调用线程不安全函数的函数

```
unsigned int next = 1;
int rand(void)
{
     next = next * 1103515245 + 12345;
     return (unsigned int) (next / 65536) % 32768;
}

void srand(unsigned int seed)
{
      next = seed;
}
```

```
int rand_r(unsigned int* nextp)
{
     *nextp = *nextp * 1103515245 + 12345;
      return (unsigned int) (*nextp / 65536) % 32768;
}
```

修正方法：不使用任何全局/静态数据，使用参数传递状态信息

缺　　点：增加程序员学习和编码负担，改变原调用程序的代码

四类不安全的线程函数

- 不保护共享变量的函数
- 保持跨越多个调用的状态函数
- 返回指向静态变量指针的函数
  - 多线程并发
  - 线程A拟读取的结果可能已被线程B改写
- 调用线程不安全函数的函数

```
#include <netdb.h>

struct hostent *gethostbyname(const char * hostname);

返回：非空指针——成功，空指针——出错，同时设置h_errno
```

```
struct hostent* gethostbyname_ts(char* host)
{
    struct hostent* shared, * unsharedp;
    unsharedp = Malloc(sizeof(struct hostent));
    P(&mutex)
    shared = gethostbyname(hostname);
    *unsharedp = * shared;
    V(&mutex);
    return unsharedp;
}
```