阅码场
YOMOCODE

多线程编程(2)

麦当劳喜欢您来，喜欢您再来

扫描关注

Linux阅码场

多线程编程(2)

2.1 信号量、互斥体
2.2 条件变量与同步
2.3 用户空间spinlock?
2.4 如何正确的加锁？
2.5 race condition的调试：ThreadSanitizer和helgrind
2.6 条件变量
2.7 pthread_mutex与优先级继承

对POSIX信号量的操作函数：

- int sem_init(sem_t *sem, int pshared, unsigned int value);
- int sem_wait(sem_t * sem);
- int sem_trywait(sem_t * sem);
- int sem_post(sem_t * sem);
- int sem_getvalue(sem_t * sem, int * sval);
- int sem_destroy(sem_t * sem);

信号量实例

T1

```
void *produce(void *arg)
{
  int i;
  for (i = 0; i < nitems; i++) {
    sem_wait(&shared.nempty);
    sem_wait(&shared.mutex);
    shared.buff[i % NBUFF] = i;
    cout << "Product " << shared.buff[i %
      NBUFF] << endl;
    sem_post(&shared.mutex);
    sem_post(&shared.nstored);
  }
  return(NULL);
}
```

T2

```
void *consume(void *arg)
{
  int i;
  for (i = 0; i < nitems; i++) {
    sem_wait(&shared.nstored);
    sem_wait(&shared.mutex);
    if (shared.buff[i % NBUFF] != i)
      cout << "buff[" << i  <<"] = " <<  shared.buff[i %
     NBUFF] << endl;
    cout << "Consumer:" << shared.buff[i % NBUFF] <<
     endl;
    sem_post(&shared.mutex);
    sem_post(&shared.nempty);
  }
  return(NULL);
}
```

信号量实例(续)

```
int main(int argc, char **argv)
{
    pthread_t tid_produce, tid_consumer;
    if (argc != 2){      cout << "Usage: prodcons number" << endl; exit(0);   }
    nitems = atoi(argv[1]);
    sem_init(&shared.mutex, 0, 1);
    sem_init(&shared.nempty, 0, NBUFF);
    sem_init(&shared.nstored, 0, 0);
    pthread_create(&tid_produce, NULL, produce, NULL);
    pthread_create(&tid_consumer, NULL, consume, NULL);
    pthread_join(tid_produce, NULL);
    pthread_join(tid_consumer, NULL);
    sem_destroy(&shared.mutex);
    sem_destroy(&shared.nempty);
    sem_destroy(&shared.nstored);
    exit(0);
}
```

```
pthread_mutex_t mutex;
pthread_mutex_init (&mutex,NULL);

pthread_mutex_lock (&mutex);
...
pthread_mutex_unlock(&mutex);
```

查看mutex

```
   deadlock.c
   34                  int tid1,tid2;
   35                  pthread_mutex_init(&mutex_1,NULL);
   36                  pthread_mutex_init(&mutex_2,NULL);
   37                  pthread_create(&tid1,NULL,child1,NULL);
   38                  pthread_create(&tid2,NULL,child2,NULL);
   39                  do{
   40                          sleep(2);
B+>41                  }while(1);
   42                  pthread_exit(0);
   43          }
   44
   45
   46
   47
   48
   49
   50
multi-thre Thread 0xb7df2 In: main                                    Line: 41   PC: 0x804872e
$1 = {__data = {__lock = 2, __count = 0, __owner = 4353, __kind = 0, __nusers = 1, {d = {__espins = 0,
        __elision = 0}, __list = {__next = 0x0}}},
  __size = "\002\000\000\000\000\000\000\000\001\021\000\000\000\000\000\000\001\000\000\000\000\000\000",
  __align = 2}
(gdb) p mutex_2
$2 = {__data = {__lock = 1, __count = 0, __owner = 4354, __kind = 0, __nusers = 1, {d = {__espins = 0,
        __elision = 0}, __list = {__next = 0x0}}},
  __size = "\001\000\000\000\000\000\000\000\002\021\000\000\000\000\000\000\001\000\000\000\000\000\000",
  __align = 1}
(gdb)
```

spin_lock

适合场景：

锁住的区间短

区间经常发生

区间可能成为性能瓶颈

锁住大区间可能导致很高的CPU利用率和性能下降


```c
#include <pthread.h>

int pthread_spin_lock(pthread_spinlock_t *lock);
int pthread_spin_trylock(pthread_spinlock_t *lock);
int pthread_spin_unlock(pthread_spinlock_t *lock);
```

正确加锁

线程安全、可重入问题

三要素

- ✓ 同一把锁
- ✓ 语义整体(事务的概念)
- ✓ 粒度最小

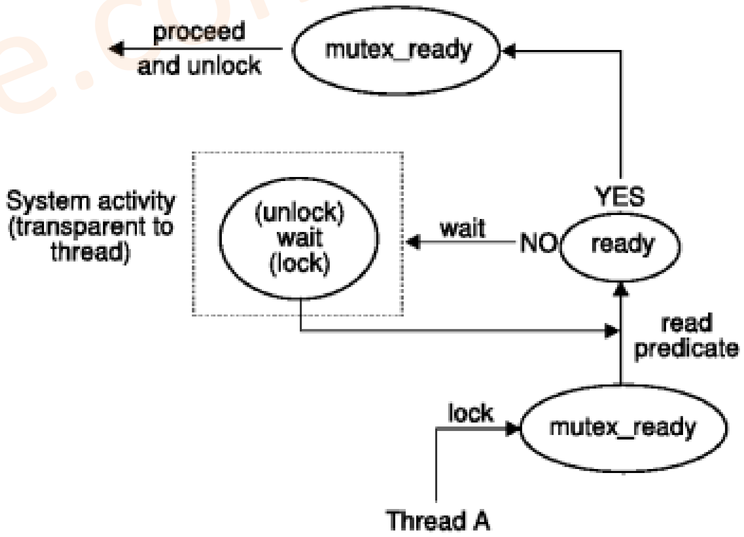
条件变量

```
pthread_mutex_t count_lock;
pthread_cond_t count_nonzero;
unsigned count;
decrement_count()
{
        pthread_mutex_lock (&count_lock);
        while(count==0)
        pthread_cond_wait( &count_nonzero, &count_lock);
        count=count -1;
        pthread_mutex_unlock (&count_lock);
}
```

T1

```
increment_count()
{
        pthread_mutex_lock(&count_lock);
        if(count==0)
                pthread_cond_signal(&count_nonzero);
        count=count+1;
        pthread_mutex_unlock(&count_lock);
}
```

T2



helgrind

Helgrind可以检测下面三类错误：

1.POSIX pthreads API的错误使用
2.由加锁和解锁顺序引起的潜在的死锁
3.数据竞态--在没有锁或者同步机制下访问内存

运行方法：

valgrind --tool=helgrind ./a.out

```
#include <pthread.h>

pthread_mutex_t mutex;

void *still_locked(void *args)
{
    (void)args;
    pthread_mutex_lock(&mutex);
    pthread_exit(0);
    return NULL;
}

int main()
{
    pthread_mutex_init(&mutex, NULL);
    pthread_t a;
    pthread_create(&a, NULL, still_locked,
NULL);
    pthread_join(a, NULL);
    return 0;
}
```

```
baohua@baohua-VirtualBox:~/develop/training/thread$ valgrind --tool=helgrind ./a.out
==17524== Helgrind, a thread error detector
==17524== Copyright (C) 2007-2013, and GNU GPL'd, by OpenWorks LLP et al.
==17524== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==17524== Command: ./a.out
==17524==
==17524== ---Thread-Announcement------------------------------------------
==17524==
==17524== Thread #2 was created
==17524==    at 0x4166298: clone (clone.S:108)
==17524==
==17524== ----------------------------------------------------------------
==17524==
==17524== Thread #2: Exiting thread still holds 1 lock
==17524==    at 0x4062014: start_thread (pthread_create.c:453)
==17524==    by 0x41662AD: clone (clone.S:129)
==17524==
==17524==
==17524== For counts of detected and suppressed errors, rerun with: -v
==17524== Use --history-level=approx or =none to gain increased speed, at
==17524== the cost of reduced accuracy of conflicting-access information
==17524== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 1 from 1)
```

ThreadSanitizer

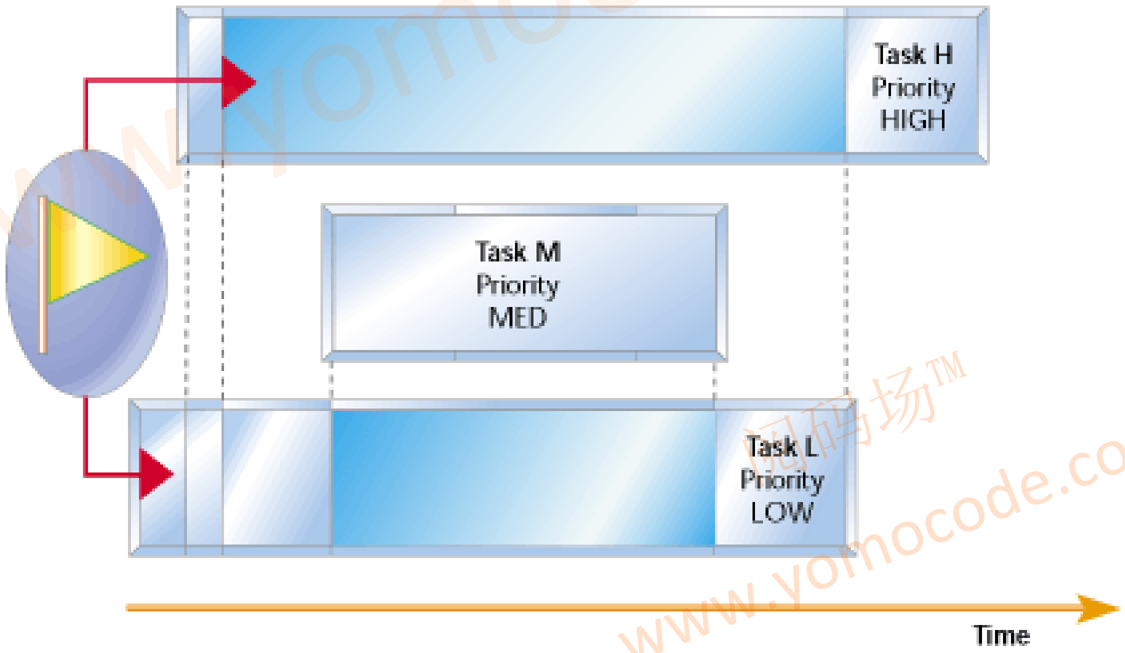
ThreadSanitizer引入编译选项-fsanitize=thread来分析data race。

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

int Global;

void *Thread1(void *x) {
  Global++;
  return NULL;
}

void *Thread2(void *x) {
  Global--;
  return NULL;
}

int main() {
  pthread_t t[2];
  pthread_create(&t[0], NULL, Thread1, NULL);
  pthread_create(&t[1], NULL, Thread2, NULL);
  pthread_join(t[0], NULL);
  pthread_join(t[1], NULL);
}
```

```
baohua@ubuntu:~/develop/training/thread$ gcc simple_race.c -fsanitize=thread -g
baohua@ubuntu:~/develop/training/thread$ ./a.out
==================
WARNING: ThreadSanitizer: data race (pid=14494)
  Read of size 4 at 0x00000060107c by thread T2:
    #0 Thread2 /home/baohua/develop/training/thread/simple_race.c:12 (a.out+0x000000400998)
    #1 <null> <null> (libtsan.so.0+0x0000000230d9)

  Previous write of size 4 at 0x00000060107c by thread T1:
    #0 Thread1 /home/baohua/develop/training/thread/simple_race.c:7 (a.out+0x00000040095b)
    #1 <null> <null> (libtsan.so.0+0x0000000230d9)

  Location is global 'Global' of size 4 at 0x00000060107c (a.out+0x00000060107c)

  Thread T2 (tid=14497, running) created by main thread at:
    #0 pthread_create <null> (libtsan.so.0+0x000000027577)
    #1 main /home/baohua/develop/training/thread/simple_race.c:19 (a.out+0x000000400a23)

  Thread T1 (tid=14496, finished) created by main thread at:
    #0 pthread_create <null> (libtsan.so.0+0x000000027577)
    #1 main /home/baohua/develop/training/thread/simple_race.c:18 (a.out+0x000000400a04)

SUMMARY: ThreadSanitizer: data race /home/baohua/develop/training/thread/simple_race.c:12 Thread2
==================
ThreadSanitizer: reported 1 warnings
```

优先级翻转

- 高优先级线程等低优先级线程释放锁的过程中，中等优先级线程打断低优先级线程



PTHREAD_PRIO_INHERIT

```
int
pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol);
```

- ✓ PTHREAD_PRIO_NONE:
- ✓ PTHREAD_PRIO_INHERIT: 优先级继承

what really happened on mars

- 探路者有一个“information bus”，总线管理任务以高优先级运行，负责在总线上放入或者取出各种数据。它被设计为最重要的任务，并且要保证能够每隔一定的时间就可以操作总线。对总线的异步访问是通过互斥锁(mutexes)来保证的。
- 另有一个气象数据搜集任务，它的运行频度不高，也只有低优先级，它只向总线写数据。写的过程是，申请/获得总线互斥量，进行写操作，完成后释放互斥量。
- 探路者上还有一个以中等优先级运行的通信任务，通信任务和总线是没有什么瓜葛的。
- 气象任务（低优先级）获得互斥量并写总线的时候，一个中断的发生导致了通信任务（中优先级）被调度并就绪，调度的时机正好是总线管理任务（高优先级）等待在总线访问互斥量上的时候。

谢 谢！

阅码场出品