# 死锁

2024年5月27日 10:45

deadlock.c

```c
#include <func.h>

typedef struct {
    int id;
    char name[25];
    int balance;
    // 细粒度锁
    pthread_mutex_t mutex;
} Account;

Account acct1 = {1, "xixi", 1000, PTHREAD_MUTEX_INITIALIZER};
Account acct2 = {2, "peanut", 100, PTHREAD_MUTEX_INITIALIZER};

int transfer(Account* acctA, Account* acctB, int money) {
    pthread_mutex_lock(&acctA->mutex);
    sleep(1);   // 增加坏的调度的概率
    pthread_mutex_lock(&acctB->mutex);

    if (acctA->balance < money) {
        pthread_mutex_unlock(&acctA->mutex);
        pthread_mutex_unlock(&acctB->mutex);

        return 0;
    }

    acctA->balance -= money;
    acctB->balance += money;

    pthread_mutex_unlock(&acctA->mutex);
    pthread_mutex_unlock(&acctB->mutex);

    return money;
}

void* start_routine1(void* args) {
    int money = (int) args;
    int ret = transfer(&acct1, &acct2, money);
    printf("%s -> %s: %d\n", acct1.name, acct2.name, ret);
    return NULL;
}

void* start_routine2(void* args) {
    int money = (int) args;
    int ret = transfer(&acct2, &acct1, money);
    printf("%s -> %s: %d\n", acct2.name, acct1.name, ret);
    return NULL;
}

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

    pthread_create(&tid1, NULL, start_routine1, (void*)900);
    pthread_create(&tid2, NULL, start_routine2, (void*)100);

    // 主线程等待子线程结束
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    printf("%s: balance = %d\n", acct1.name, acct1.balance);
    printf("%s: balance = %d\n", acct2.name, acct2.balance);
    return 0;
}
```

acct1 → acct2　　acct2 → acct1

tid2　　tid1　　tid2

拿 acct1->mutex

→ 切换

拿 acct2->mutex

切换

拿 acct2->mutex

（阻塞）

拿 acct1->mutex

（阻塞）

子线程

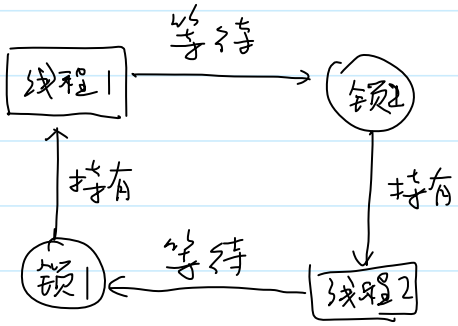
```
he@he-vm:~$ ps -elLf | grep "./deadlock"
0 S he          3472    2656    3472  0    3  80   0 -  4792 futex_ 11:02 pts/1    00:00:00 ./deadlock
1 S he          3472    2656    3473  0    3  80   0 -  4792 futex_ 11:02 pts/1    00:00:00 ./deadlock
1 S he          3472    2656    3474  0    3  80   0 -  4792 futex_ 11:02 pts/1    00:00:00 ./deadlock
```

# 死锁的解决方案

2024年5月27日 11:10

① 互斥
② 持有并等待
③ 不能抢占
④ 循环等待

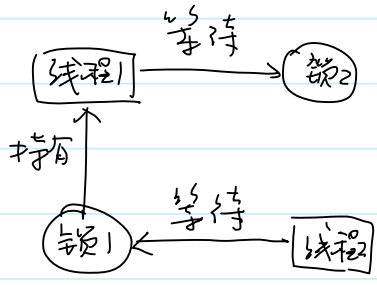
同时成立 ⟹ 死锁 (×)




④ ~~循环等待~~

按固定的顺序，依次获取锁 (√)

```
// 按id的顺序依次获取锁
if (acctA->id < acctB->id) {
    pthread_mutex_lock(&acctA->mutex);
    sleep(1);   // 增加坏的调度的概率
    pthread_mutex_lock(&acctB->mutex);
} else {
    pthread_mutex_lock(&acctB->mutex);
    sleep(1);   // 增加坏的调度的概率
    pthread_mutex_lock(&acctA->mutex);
}
```




③ ~~不能抢占~~

获取锁1

sleep(1)

获取锁2

sleep(1)

sleep(1)
尝试获取锁 2

sleep(1)
尝试获取锁 1

```
28      // 3. 不能抢占
29  start:
30      pthread_mutex_lock(&acctA->mutex);
31      sleep(1);
32      int err = pthread_mutex_trylock(&acctB->mutex);
33      if (err) {
34          // 主动释放获取的锁
35          pthread_mutex_unlock(&acctA->mutex);
36          int seconds = rand() % 10;
37          sleep(seconds);
38          goto start;
39      }
```

② 持有并等待

```
// 2. 持有并等待
pthread_mutex_lock(&protection);
pthread_mutex_lock(&acctA->mutex);
sleep(1);
pthread_mutex_lock(&acctB->mutex);
pthread_mutex_unlock(&protection);
```

① 互斥 → 复杂的指令

CAS 操作 (CPU指令)

Compare - And - Swap

Lock - free 算法

wait - free 算法

⇒ 互斥

# 等待条件成立

2024年5月27日 14:26

## 条件变量（pthread_cond_t）

- 条件变量：总是提供了一个等待、唤醒机制
- 至于条件何时成立，何时不成立，取决于业务。

### #1. 初始化

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

attr → 一般填NULL，表示默认属性。

PTHREAD_COND_INITIALIZER → 默认属性

### #2. 当条件不成立，等待。

语义：当wait返回时，条件曾经成立过。（现在还成立吗？不知道，还需检测）

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

abstime ⇒ 超时时间

wait ⇒ 无限期等待

问题：pthread_cond_wait() 为什么需要传递互斥锁。

互斥锁：保护 cond 变量、共享资源

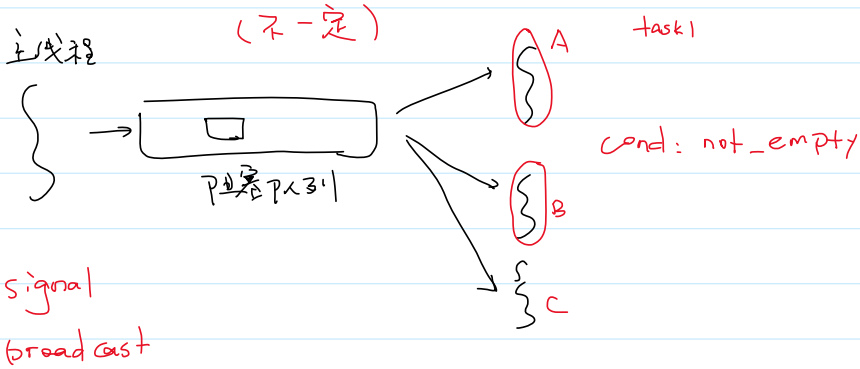
→ 多个线程共享

① 释放 mutex
② 陷入阻塞状态
} 原子操作

③ 当返回时，该线程一定再一次获取了 mutex。

Q：pthread_cond_wait 返回时，条件一定成立吗？

（不一定）



#3. 当条件成立时，唤醒等待的线程。

至少会唤醒一个

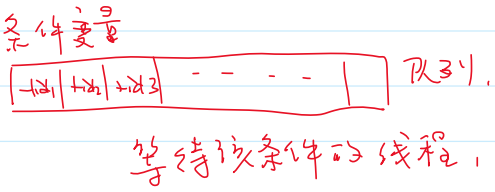
```
int pthread_cond_signal(pthread_cond_t *cond);
```

唤醒是一个等待该条件变量的线程。

注意：在实际实现时，为了性能，一个也可能唤醒多个线程

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

唤醒是所有等待该条件变量的线程。

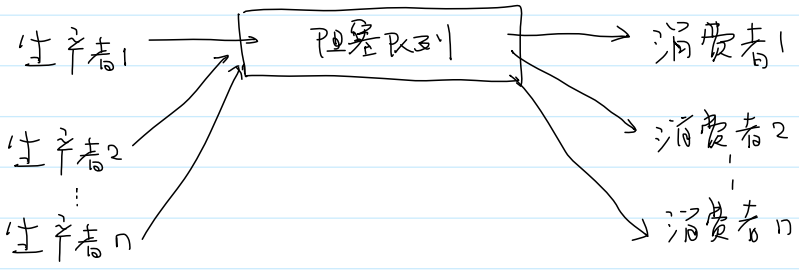



#4. 销毁

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

# 生产者消费者模型

2024年5月27日 14:54




阻塞队列：当队列满时，如果线程往阻塞队列中添加东西，线程会陷入阻塞

当队列空时，如果线程往阻塞队列中取东西，线程会陷入阻塞

生产者：生产"商品"，

如果队列满了，生产者陷入阻塞 等待队列不满 (Not_full)

如果队列不满，将商品添加到阻塞队列；队列非空 (not_empty)，唤醒消费者

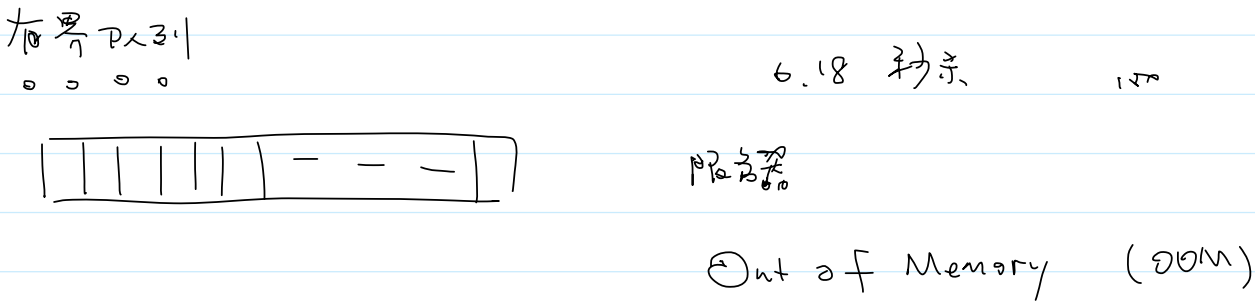
消费者：消费"商品"

如果队列空了，消费者陷入阻塞 等待队列非空 (Not_empty)

如果队列非空，从阻塞队列中获取商品；队列不满 (not_full)，唤醒生产者

# 阻塞队列

2024年5月27日 15:05




```
 8 typedef int E;
 9
10 // 循环数组实现队列
11 typedef struct {
12     E elements[N];
13     int front;
14     int rear;
15     int size;
16
17     pthread_mutex_t mutex;
18     pthread_cond_t  not_empty;  // 非空
19     pthread_cond_t  not_full;   // 不满
20 } BlockQ;
21
22 // API
23 void blockq_create(void);
24 void blockq_destroy(BlockQ* q);
25
26 void blockq_push(E val);
27 E    blockq_pop(BlockQ* q);
28 E    blockq_peek(BlockQ* q);
29 bool blockq_empty(BlockQ* q);
30 bool blockq_full(BlockQ* q);
```

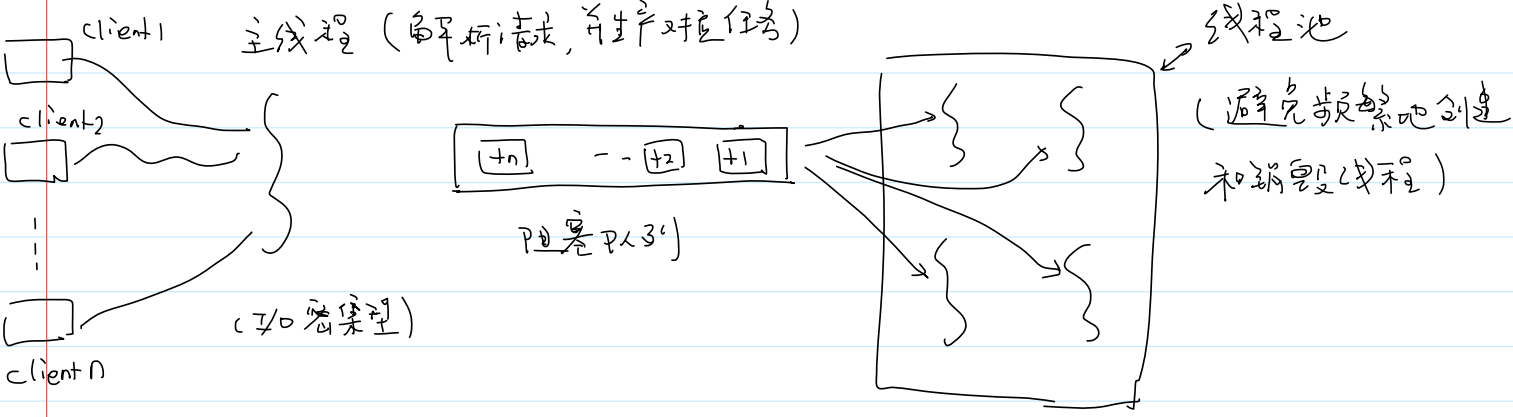
BlockQ.c buffer

```c
#include "BlockQ.h"

// 创建空的阻塞队列
BlockQ* blockq_create(void) {
    BlockQ* q = (BlockQ*) calloc(1, sizeof(BlockQ));

    pthread_mutex_init(&q->mutex, NULL);
    pthread_cond_init(&q->not_empty, NULL);
    pthread_cond_init(&q->not_full, NULL);

    return q;
}

void blockq_destroy(BlockQ* q) {
    pthread_mutex_destroy(&q->mutex);
    pthread_cond_destroy(&q->not_empty);
    pthread_cond_destroy(&q->not_full);

    free(q);
}

void blockq_push(BlockQ* q, E val) {
    // 获取锁
    pthread_mutex_lock(&q->mutex);
    // 注意事项：一定要写成 while
    while (q->size == N) {
        // 1. 释放q->mutex
        // 2. 陷入阻塞状态
        // 3. 当pthread_cond_wait返回时，一定再一次获取了q->mutex
        // 语义：返回时，条件曾经成立过，现在是否成立，不确定；需要再一次检查
        // 存在虚假唤醒现象
        pthread_cond_wait(&q->not_full, &q->mutex);
    } // a. 获取了q->mutex; b. q->size != N

    q->elements[q->rear] = val;
    q->rear = (q->rear + 1) % N;
    q->size++;
    // not_empty条件成立，唤醒等待not_empty条件的线程
    pthread_cond_signal(&q->not_empty);

    pthread_mutex_unlock(&q->mutex);
}

E blockq_pop(BlockQ* q) {
    pthread_mutex_lock(&q->mutex);
    while (q->size == 0) {
        pthread_cond_wait(&q->not_empty, &q->mutex);
    }
    // a. 获取了q->mutex; b. q->size != 0
    E retval = q->elements[q->front];
```

```
51      q->front = (q->front + 1) % N;
52      q->size--;
53
54      // not_full条件成立，唤醒等待not_full条件的线程
55      pthread_cond_signal(&q->not_full);
56
57      pthread_mutex_unlock(&q->mutex);
58
59      return retval;
60  }
61
62  E blockq_peek(BlockQ* q) {
63      pthread_mutex_lock(&q->mutex);
64      while (q->size == 0) {
65          pthread_cond_wait(&q->not_empty, &q->mutex);
66      }
67      // a. 获取了q->mutex; b. q->size != 0;
68      E retval = q->elements[q->front];
69      pthread_mutex_unlock(&q->mutex);
70      return retval;
71  }
72
73  bool blockq_full(BlockQ* q) {
74      pthread_mutex_lock(&q->mutex);
75      int size = q->size;
76      pthread_mutex_unlock(&q->mutex);
77      return size == N;
78  }
79
80  bool blockq_empty(BlockQ* q) {
81      pthread_mutex_lock(&q->mutex);
82      int size = q->size;
83      pthread_mutex_unlock(&q->mutex);
84      return size == 0;
85  }
```

# 实现生产者消费者模型

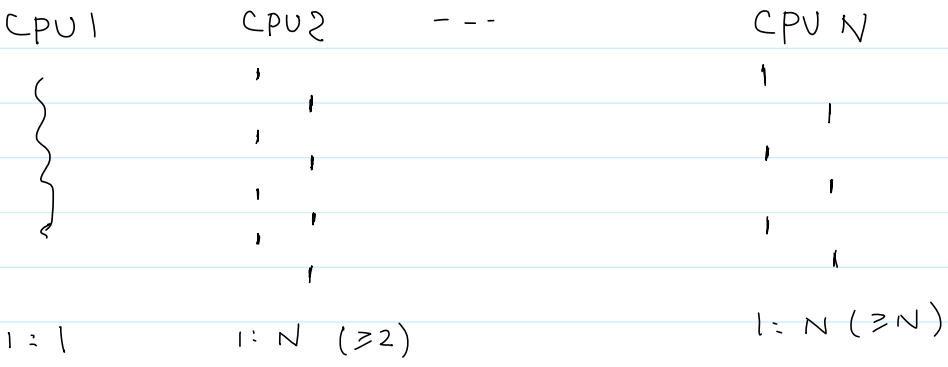
2024年5月27日 16:09




1. 应用程序，应该包含多个线程（包括 main 线程）

CPU的核数

任务的负载 → I/O密集型任务
→ 计算密集型任务



pc.c

```c
#include <func.h>
#include "BlockQ.h"

typedef struct {
    pthread_t* threads;
    int nums;   // 线程数目
    BlockQ* q;
} ThreadPool;

void* start_routine(void* args) {
    ThreadPool* pool = (ThreadPool*) args;
    pthread_t tid = pthread_self();

    for (;;) {
        // 从阻塞队列中获取任务
        E task_id = blockq_pop(pool->q);
        if (task_id == -1) {
            pthread_exit(NULL);
        }
        printf("%lx: execute task %d\n", tid, task_id);
        sleep(3);   // 模拟执行任务
        printf("%lx: task %d done\n", tid, task_id);
    }

    return NULL;
}

ThreadPool* threadpool_create(int n) {
    ThreadPool* pool = (ThreadPool*) malloc(sizeof(ThreadPool));

    pool->threads = (pthread_t*) malloc(n * sizeof(pthread_t));
    pool->nums = n;
    pool->q = blockq_create();
    // 创建线程
    for(int i = 0; i < n; i++) {
        pthread_create(pool->threads + i, NULL, start_routine, (void*)pool);
    }

    return pool;
}

void threadpool_destroy(ThreadPool* pool) {
    blockq_destroy(pool->q);
    free(pool->threads);
    free(pool);
}

int main(int argc, char* argv[])
{
    // 1. 创建线程池, 并初始化
    ThreadPool* pool = threadpool_create(8);

    // 2. 主线程生产任务
    for(int i = 0; i < 100; i++) {
        blockq_push(pool->q, i + 1);
    }

    sleep(5);

    // 3.a 暴力退出线程池
    /* for (int i = 0; i < 8; i++) { */
    /*     pthread_cancel(pool->threads[i]); */
    /* } */

    // 3.b 优雅退出线程池
    for (int i = 0; i < pool->nums; i++) {
        blockq_push(pool->q, -1); // -1表示退出任务
    }

    for(int i = 0; i < pool->nums; i++) {
        pthread_join(pool->threads[i], NULL);
    }

    // 4. 销毁线程池
    threadpool_destroy(pool);
    return 0;
```

```
76      return 0;
77  }
78
```