221900175 毛九弢 期中作业

1. 使用条件变量需要配合wait和signal原语。一种更通用的同步形式是只用一条同步原语waituntil，它以任意的布尔谓词作为参数。例如： waituntil x < 0 or y+z < n。 其意义为等待直到所需要的条件达成，这样就不需要signal原语，也更加通用。然而其并没有被现代操作系统所采用。尝试回答可能的原因是什么?

> 参考 **OSTEP** 第30章 和 **PPT** 6 同步-基础 生产者-消费者问题 回答:

`waituntil` 是一种基于**自旋**实现的同步,
也就是waituntil的线程一直在"轮询"条件是否满足.

`wait(c,lk) & signal(c)` 是一种基于**阻塞**实现的同步,
即wait的线程在判断条件不满足时,**阻塞睡眠,并释放锁**,
等到某个线程,调用了signal(c),会向wait的线程发送信号,**唤醒并持有锁**.

`waituntil` 有一个变量处于临界区,
如果不加锁,有安全问题! 很大可能导致错误!
需要加锁之后,再进行自旋,这样一来效率极低!

`wait(c,lk) & signal(c)` 就能非常好地解决这个性能问题.

2.互斥锁的两种实现：不断自旋（spin）和基于阻塞（block），什么情况下用spin好一点，什么时候基于阻塞的实现好一点?

首先进行对两个锁的特征分析

自旋锁

1. 其优点
   - 在等待期间会一直检查锁是否可用，不会让出 CPU，因此在短时间内能够获得锁的概率更高
2. 其代价
   - 除了进入入临界区的线程，其他处理器上的线程都在空转
   - 如果临界区执行时间过长，其他线程浪费的CPU越多
   - 此外，如果发生中断将临界区的线程切出去了，计算资源浪费更加严重

阻塞锁

1. 优点
   - 使得线程在等待锁时进入阻塞状态,不会导致忙等待,让出消耗 CPU 资源.
2. 其代价
   - 需要陷入系统内核,有额外的开销

所以,我们可以总结出:

1. **自旋锁**的适合情况是
   - 适合于短期的争用情况,锁被持有的时间很短,等待时间很短.
   - 线程数量不多,线程间竞争不激烈
   - 系统中断频次较少.
   - 多核处理器.
2. **阻塞锁**的适合情况是
   - 锁被持有的时间很长,等待时间很长.
   - 线程较多,多个线程等待一把锁的情况频发.
   - 系统中断频发.
   - 系统的负载较高, CPU资源不足.
   - 单核处理器.

---

3. 考虑可以购买一台每秒执行五亿条指令的单处理器，或者购买一台拥有8个核心的多处理器，其中每个核心每秒执行一亿条指令。根据阿姆达尔定律(Amdahl's Law)，解释什么样的程序应该使用第一种CPU，什么样的程序应该使用第二种CPU。

$Amdahl's Law$：

$$\text{Speedup}(S) = \frac{1}{(1-P) + \frac{P}{N}}$$

其中：

- S 表示并行计算后的加速比（Speedup）。
- P 表示并行化部分所占的比例，即可以并行执行的部分所占的比例。
- N 表示处理器的数量。

阿姆达尔定律说明了并发系统性能的提升受到两个因素的限制：

1. **并行化部分的比例** P：即可以并行执行的部分在整体任务中所占的比例。如果 P 较小，表示任务中存在大量的串行部分，那么无论增加多少处理器，总体加速比也会受到限制。
2. **处理器的数量** N：增加处理器数量可以提高并行计算的效率，但是也会受到串行部分的限制。

故而:

1. 适合选择每秒五亿条指令的单处理器的程序的特质:
   - 大部分为串行计算的程序,因为并行化任务对于速度提升没有太多影响
   - 如:单线程应用程序或者难以并行化的程序
2. 适合选择8核心每秒一亿条指令多处理器的程序的特质:
   - 大部分任务可被分解成并行任务，并可以充分利用多核处理器的并行执行能力
   - 如:大规模数据处理,模型训练/炼丹(不过好像是用gpu(楽))

---

4. 考虑如下哲学家就餐问题的变种：所有筷子都放在桌子的中间，哲学家吃饭的时候只要从中任意拿两个筷子即可，吃完了再放回中间。有一种避免死锁的方式是提供足够多的资源。那么如果有N个哲学家，至少要提供多少筷子即可一定避免死锁问题？为什么?

跟据抽屉原理,至少放N+1只筷子就能满足;
因为最坏情况是每个人都拿了一只筷子,也就需要N只筷子,
那么无论谁再拿一只筷子,就能吃饭并释放筷子,
这样又有两位哲学家能拿到筷子并吃饭释放筷子了.

---

5. 对于一个真实的计算机系统，可用的资源和进程的资源不再一成不变，而是在动态的变化。资源会损害和替换、新的进程也来来去去，新的资源会被购买并加入到系统。如果采用银行家算法控制死锁，那么下面哪些变化是安全的（不会导致死锁），哪些是变化可能导致不安全，为什么?

a. 增加可用资源 （新的资源加入到系统）
b. 减少可用资源 （资源被系统永久移除）
c. 增加一个进程的最大需求
d. 减少一个进程的最大需求
e. 增加进程的数量
f. 减少进程的数量

a. 安全, 因为在"更穷"的时候,都能完成所有进程,那"富裕"的时候,按照原先的计划也是能完成的
b. 不安全, 之前的计划可能是当时唯一一条能够"苟活"到结束的计划安排,现在"更贫穷"了,提供的资源变少了,计划中有一环可能就满足不了了,造成死锁.
c. 不安全,之前的计划可能是当时唯一一条能够"苟活"到结束的计划安排,现在计划中有一环的需要的资源变多了,计划中有一环可能就满足不了了,造成死锁.
d. 安全, 按照原先的计划就能完成
e. 不安全,可能集齐系统所有资源都满足不了这个进程的需求,
f. 安全, 这个操作等价于将减少的进程的所有需求设为0,并将减少的进程的目前拥有的资源增加到系统的可用资源上

---

6. 下面的代码展示了一个栈的实现

```
typedef struct __Node {
     struct __Node *next;
     int value;
} Node;

void push(Node **top_pter, Node *n){
     n->next = *top_ptr;
     *top_ptr = n;
}

Node *pop(Node **top_ptr){
     if (*top_ptr == NULL)
          *return NULL;
     Node *p = *top;
     *top_ptr = (*top_ptr)->next;
     return p;
}
```

a) 请分析该实现是否线程安全并说明原因

b) 请尝试使用互斥锁来保证该代码片段的安全

c) 除了基本的push和pop操作以外，栈通常还提供一个top操作，用于查询栈顶元素的值。该查询不会移出该元素（与pop不同）。如果我们实现该查询和上述代码片段中pop函数相似（除了不改变top_ptr指针），实现过程也不用互斥锁，而是直接读取top的指针的值，那么该实现会出现错误吗?

d) 如果需要同步原语来保护top操作，用什么比较好？为什么?

1. 不安全,因为如果是多线程任务,多个线程可以同时进入临界区,没有上锁
2. 实现

```c
#include <pthread.h>

pthread_mutex_t mutex_lock = PTHREAD_MUTEX_INITIALIZER;

typedef struct __Node
{
    struct __Node *next;
    int value;
} Node;

void push(Node **top_ptr, Node *n)
{
    pthread_mutex_lock(&mutex_lock);
    n->next = *top_ptr;
    *top_ptr = n;
    pthread_mutex_unlock(&mutex_lock);
}

Node *pop(Node **top_ptr)
{
    pthread_mutex_lock(&mutex_lock);
    Node * toPop = NULL;
    if(*top_ptr != NULL){
        toPop = (*top_ptr);
        *top_ptr = toPop->next;
    }
    pthread_mutex_unlock(&mutex_lock);
    return toPop;
}
```

3. 由于top的只读性,所以写不会出错,但是读会出错.
   考虑下面的情况:

```
stack   : [Node1]
thread1 : toPop = (*top_ptr);
thread2 : if(*top_ptr != NULL)      // True, so except not a NULL
thread1 : *top_ptr = toPop->next;   // *top_ptr is NULL
thread2 : return (*top_ptr);        // return a NULL
```

4. 所以考虑使用读写锁,并根据使用的情况,选择读者优先，或写者优先，或公平排队.
   原因是，这是一个明显的读者写者问题，难道不应该使用读写锁吗？（菜）
   因为这样的话，读者们相互不互斥，写者们和读者写者互斥，保全安全性的同时，也保证了性能。

---

7. 考虑我们给出了一种n个线程互斥的实现Flaky Lock，代码如下（注：假设我们是SC的内存模型，且编译器没有乱序优化）

```c
#include <stdbool.h>
// 假设 ThreadID.get() 返回当前线程的 ID
int ThreadID_get() {
 // 实现获取线程 ID 的代码
}
typedef struct {
    int turn;
    bool busy;
} Flaky;
void Flaky_init(Flaky *lock) {
    lock->turn = 0;
    lock->busy = false;
}
void Flaky_lock(Flaky *lock) {
    int me = ThreadID_get();
    do {
        do {
            lock->turn = me;
        } while (lock->busy);
        lock->busy = true;
    } while (lock->turn != me);
}
void Flaky_unlock(Flaky *lock) {
    lock->busy = false;
}
```

a) 这个实现是否满足互斥性?

b) 这个实现是否无饥饿?

c) 这个实现是否无死锁? 如果没有死锁请解释为什么，如果有死锁，同样给出解释，并给出一个改进版本，使其不会出现死锁

1. 互斥满足

不可能有两个线程同时在临界区内，也就是在两次锁释放之间只有一个线程出outer loop.

证明如下:

假设存在.我们反推, 不妨设 `thread0` , `thread1` 中, `thread0` 先出 outerloop.

很trivial的一个结论是, `thread1` 要出outerloop前,得先出innerloop,
而由于 `lock->busy = false` 是出 innerloop 的条件,
所以, `thread1` 想要出 innerloop, 得在 `thread0` 执行 `lock->busy = true` 前, 出 innerloop,
但 `thread1` 出了innerloop, 就代表了他执行了 `lock->turn = me` .
但是这样的话, `thread0` 就出不去 outerloop 了
那就和两个thread都出 outerloop 矛盾了,所以假设不成立.

2. 有饥饿

我们希望讨论,在不死锁的情况下的饥饿状态.
从代码来看, 这个算法, 显然有可能出现违背有界等待的情况.
但是我发现如果产生了饥饿状态, 下一个状态很大可能是死锁.
在这个算法中, 内部循环的线程会频繁修改 lock->turn,
让出了inner loop的线程 出不去outer loop, 会产生饥饿,
而这种情况的一种分类, 会直接导致死锁.

3. 死锁的可能很大

我们发现死锁的条件在于:
未获得锁的线程全部在inner loop中,且lock->busy为true.
也就是 在某一次 释放锁后, 所有出了inner loop的线程,
在outer loop的 while 判断中全部为true, 再次进入了 outer loop,
从而进入inner loop.
例子如下:

```
thread1 : lock->busy = false;       // t1 release the lock
thread2 : while (lock->busy)        // False, t2 breaks inner loop
thread3 : lock->turn = me;
thread2 : lock->busy = true;
thread3 : while (lock->busy);       // True, t3 continues inner loop
thread2 : while (lock->turn != me); // True, t2 continues big loop
thread2 : lock->turn = me;          // thread2 enters inner loop
// no thread gets the lock. 寄!
```

4. 改进如下

```c
#include <stdbool.h>
#include <stdatomic.h>
// 假设 ThreadID.get() 返回当前线程的 ID
int ThreadID_get()
{
    // 实现获取线程 ID 的代码
}
typedef struct
{
    int turn;
    bool busy;
} Flaky;
void Flaky_init(Flaky *lock)
{
    lock->turn = 0;
    lock->busy = false;
}
void Flaky_lock(Flaky *lock)
{
    int me = ThreadID_get();
    do
    {
         // 这里的while循环事实上是可以不要的啊?
         // 我没太理解原来这个锁为什么要多此一举?
        while (atomic_exchange(&lock->turn, me) != me);
    } while (__sync_val_compare_and_swap(&lock->busy,false,true));
}
void Flaky_unlock(Flaky *lock)
{
    atomic_exchange(&lock->busy, false);
}
```

---

8. 为什么在用户态不能关中断，给出两个理由

- 临界区的代码死循环了会导致整个系统也卡死了
- 中断关闭时间过长会导致很多其他重要的外界响应丢失
  （比如错过了磁盘I/O的完成事件）
- 多处理器无效