Operating Systems* Homework LaTeX

*Teacher: Shuyu Shi. TA: Gravity

1st **张逸凯** 171840708 (转专业到计科, 非重修)
Department of Computer Science and Technology
Nanjing University
zykhelloha@gmail.com

1. 应用题

3. 解:

由P, V操作的定义可以知道:

在执行了$V(S1)$之后, 信号量$S1$的值变为1, 之后执行$z = y + 1 = 5$, 在$P(S2)$操作时被阻塞.

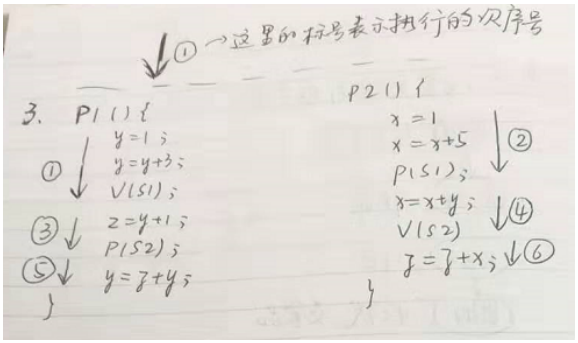
再看P2(), 因为刚刚执行了$V(S1)$, 所以$P(S1)$执行后S1变为0, 由于z = y + 1; 与 x = x + y; 的结果不相关;

所以x = x + y = 6 + 4 = 10; 然后$V(S2)$, $P(S2)$得以执行, 但$V(S2)$, $P(S2)$以下的语句相关有相互影响, 需要分类讨论:

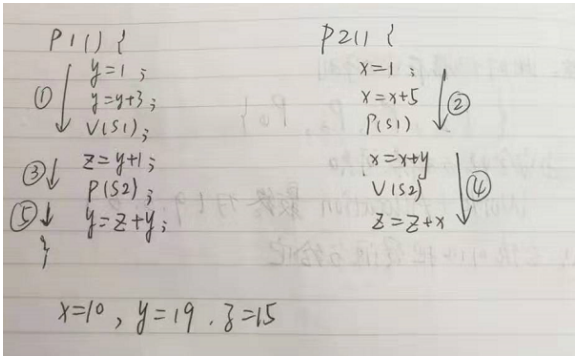
先执行$P1()$中的语句, 则最后结果为: $x = 10, y = 9, z = 15$;

先执行$P2()$中的语句, 则最后结果为: $x = 10, y = 19, z = 15$;

执行顺序如下所示(注意$V(S2)$, $P(S2)$以下顺序不同才对结果有影响):



(a) 顺序1



(b) 顺序2

*谢谢老师和助教的耐心批改.

概念题

17.解:

进程和管程的区别:

进程：进程是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。它是操作系统动态执行的基本单元，在传统的操作系统中，进程既是基本的分配单元，也是基本的执行单元。管程：管程定义了一个数据结构和能为并发进程所执行的一组操作，这组操作能同步进程和改变管程中的数据。

进程要访问临界资源时，都必须经过管程才能进入管程访问共享数据, 每次仅允许一个进程在管程内执行某个内部过程.

24. 27. 解:

简述死锁 饥饿, 银行家算法, 并举一个现实的例子.

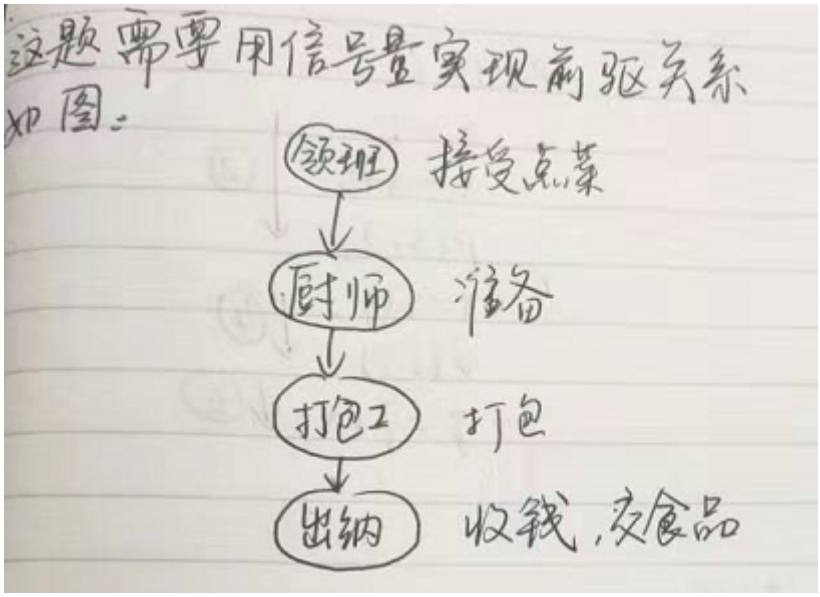
银行家算法是死锁避免算法, `os`被看成银行家, `os`管理的资源想到与银行家管理的资金, 进程向OS请求分配资源相当于用户向银行家贷款. 申请资源时, 需要测试该进程对资源的最大需求量, 如果系统现存的资源可以满足它的最大需求量则按当前的申请量分配资源, 否则就推迟分配. 进程在执行中继续申请资源的时候, 先测试该进程已占用的资源数与本次申请的资源数之和是否超过了该进程对资源的最大需求量, 若超过则拒绝分配, 若没有超过则再测试系统现存的资源能否满足该进程尚需的最大资源量, 若满足则分配, 否则也要推迟分类.

死锁是指多个进程因竞争资源而造成的一种僵局(相互等待), 若无外力作用, 这些进程都将无法向前推进; 饥饿指的是某个进程长时间不能被分配到系统不能保证某个进程的等待时间上界,从而使该*进程*长时间等待

例子可用接下来应用题的内容, 有一座独木桥, 只能容纳一辆汽车通行, 如果西侧的汽车占了桥的一半, 东侧的汽车占了桥的另一半, 两辆车都想对方让出桥(资源), 此时产生了死锁, 两辆汽车都无法过桥.

应用题

9.解:



设置五个信号量:`a1, a2, a3`

```
semaphore a1 = a2 = a3 = a4 = a5 = 0;
Foreman() {
    接受点菜...

    V(a1);  // 点菜完成
}

Cook() {
    P(a1);
    做菜...
    V(a2);  // 做菜完成
}

Package {
    P(a2);
    打包...
    V(a3);
}

Cashier {
    P(a3);
    收钱，交给顾客食物...
}
```

23.解

(1)

$$Need = \begin{pmatrix} 2 & 2 & 2 \\ 1 & 0 & 2 \\ 1 & 0 & 3 \\ 4 & 2 & 0 \end{pmatrix} = Cki - Aki$$

其中矩阵的行向量v_i表示第i个进程. ($i \in \{1, 2, 3, 4\}$)

(2)

处于安全状态, 因为系统能按照某种进程推进顺序: (P_2, P_1, P_3, P_4)执行, 而且每个进程都可以获得所需要资源. 如上是一种安全序列, 所以出于安全状态.

(3)

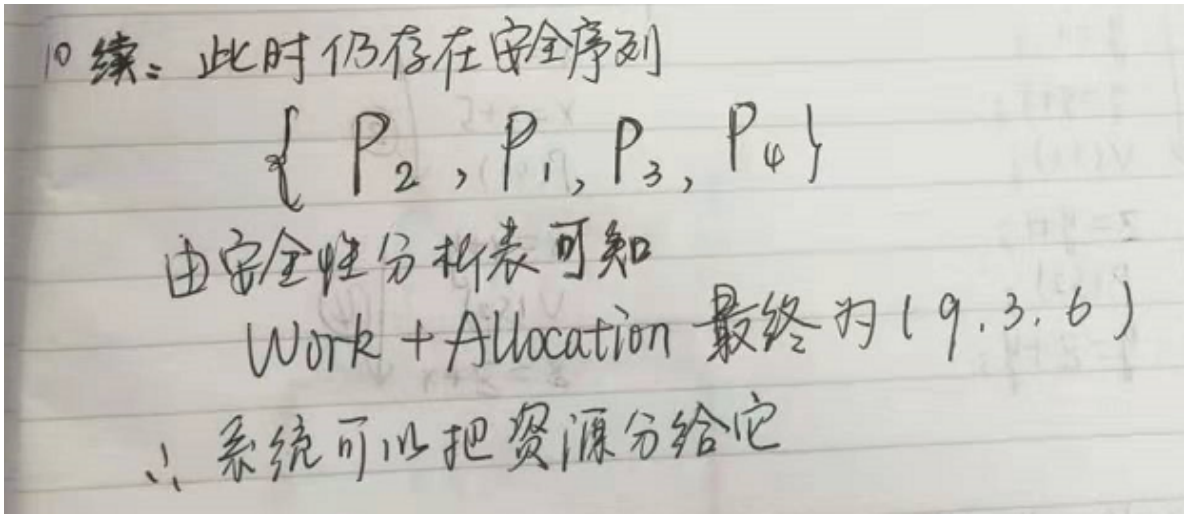
能, 按照银行家算法进行检测:

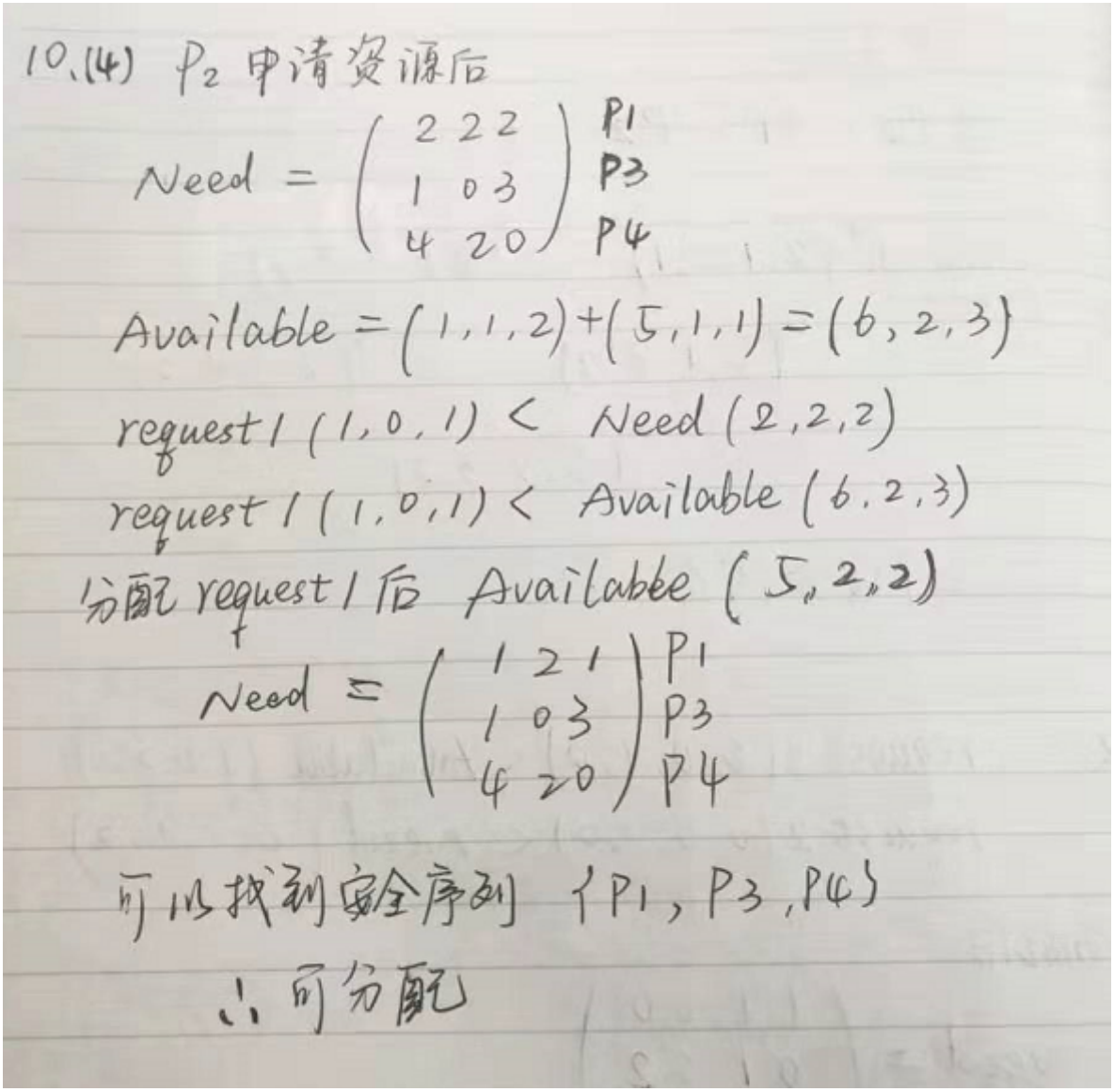
$$request2(1,0,1) < Need(1,0,2); request2(1,0,1) < Available(1,1,2)$$

$Need$ Update, $Available$ Update:

$$Need = \begin{pmatrix} 2 & 2 & 2 \\ 0 & 0 & 1 \\ 1 & 0 & 3 \\ 4 & 2 & 0 \end{pmatrix}$$

$$Available = (0,1,1)$$





31.解

不妨设东边的和西边的1车辆分别为`car1`和`car2`:

```
semaphore mutex = 1;
// 这是最基本的进程互斥的信号量模型了
car1() {
    P(mutex);
    car1 passes over the bridge...
    V(mutex);
}

car2() {
    P(mutex);
    car2 passes over the bridge...
    V(mutex);
}
```

32.解(附加)

在31的基础上将mutex 互斥信号量初始化改为 `k` 即可:

`semaphore mutex = k;`

33.解(附加)

不妨设东边的和西边的车辆分别为`car1`和`car2`:

```
semaphore a = 1, b = 0;
car1() {
    P(a);
    car1 passes over the bridge...
    V(b);
}

car2() {
    P(b);
    car2 passes over the bridge...
    V(a);
}
```

32 和 33 是 31 的进化, 需要一些简单的技巧.

47.解

(1)

```
semaphore mutex = 1;         // 储存柜互斥访问
semaphore empty = N;         // 空闲槽个数.
semaphore full = 0;          // 不空闲槽个数.

semaphore wheel = 0;         // 加工好的车轮数
semaphore frame = 0;         // 加工好的车架数

group1() {
    while(true) {
        P(empty);
        P(mutex);
        加工车架...
        V(mutex);
        V(full);    // 先申请了空闲位置之后才能放.
        V(frame);
    }
}

group2() {
    while(true) {
        for (int i = 0; i < 4; ++i)
            P(empty);
        P(mutex);
        加工车轮...
        V(mutex);
```

```
        for (int i = 0; i < 4; ++i) {
            V(full);
            V(whell);
        }
    }
}

group3() {
    while(true) {
        // 取车轮
        for (int i = 0; i < 4; ++i) {
            P(full);
            P(whell);
        }
        // 取车框
        P(full);
        P(frame);

        P(mutex);
        取并组装小汽车...
        V(mutex);

        for (int i = 0; i < 5; ++i)
            V(empty);
    }
}
```