به نام خدا

سوال اول:

الف) اگر دو برنامه ای را تصور کنیم که یکی وقتی تابع add صدا زده می شود به آن حساب مشترک مبلغی اضافه شود و برنامه دیگر وقتی تابع withdraw صدا زده می شود از آن حساب مشترک مبلغی بردارد:

```
Void add(int amount){
        while (true) {
                while (amount == BUFFER_SIZE))
                ; /* do nothing */
                buffer[in] = amount;
                in = (in + 1) % BUFFER_SIZE;
                amount += amount;
        }
}

Void withdraw(){
        item withdrawn_amount;
        while (true) {
                while (amount == 0)
                        ;/* do nothing */
                        withdrawn _amount = buffer[out];
                        out = (out + 1) % BUFFER_SIZE;
                        amount -= amount;
                }
        }
}
```

متغیر amount برای فهمیدن تغییراتی که روی amount ایجاد می شود گذاشته شده است. از آنجا که این دو برنامه به صورت هم روند پیش می روند در هنگامی که مبلغی را اضافه می کنیم و سپس همان مبلغ را بر می داریم، و انتظار داریم مبلغ دست نخورده باقی بماند ولی این اتفاق نمی افتد که در ادامه توضیح خواهم داد:

در هنگام add:

1. reg1 = amount
2. reg1 = amount + amount
3. amount = reg1

در هنگام withdraw:

1. reg2 = amount
2. reg2 = amount - amount
3. amount = amount

اگر در فرآیند add ابتدا خط اول و دوم اجرا شده و مبلغ اضافه شده در رجیستر ۱ ریخته شود و سپس به فرآیند withdraw سوئیچ کنیم و دو خط اول آن هم اجرا کنیم و مبلغ کاسته شده در رجیستر ۲ بریزیم، حال دوباره در فرآیند اول خط سوم را اجرا کنیم، مقدار amount برابر با رجیستر ۱ می شود که مبلغ اضافه شده به مبلغ قبلی بود. ولی با سوئیچ دوباره روی فرآیند دوم مقدار amount برابر با مقدار مبلغ کاسته شده از مبلغ قبلی می شود.
یعنی counter دو مقدار متفاوت از هم گرفت که این از عواقب race condtion به وجود آمد.

راه حل ها باید دارای سه ویژگی باشد:
۱) انحصار متقابل: فقط یک فرآیند می تواند وارد ناحیه بحرانی شود
۲) پیشرفت : بالاخره یکی از فرآیندها پیشرفت پیدا کند
۳) انتظار محدود: زمانی که طول می کشد تا درخواست برای ورود به ناحیه بحرانی اجابت شود کران دار باشد. برای فرآیندی قطحی پیش نیاید.

راه های متعددی برای رفع این مشکل وجود دارد از جمله الگوریتم Peterson، روش های سخت افزاری (۱. Test_and_set ۲. Compare_and_swap)، راه حل توسط os (mutex lock) و semaphore (برخلاف mutex ، cpu را بیهوده مشغول نگه نمی دارد)

سوال دوم: الف)

Subject:
Year: Month: Day: ( )

P_i	P_j
1 flag[i] = true;	2 flag[j] = true;
2 turn = j;	4 turn = i;
3 while (flag[j] && turn == j);	while (flag[i] && turn == i);
// CS	// CS
5 flag[i] = false;	flag[j] = false;

انحصار متقابل :

از آنجا که turn = i یا turn = j هردو شرط while نمی‌توانند نادرست باشند که هردو وارد CS شوند درنتیجه انحصار متقابل دارد

پیشرفت :

فرض کنیم P_j اصلا به خط flag[j] = true نرسیده باشد و قصد ورود به CS هم نداشته باشد در این صورت از P_i خط اول و دوم اجرا شده و در while گیر می‌کند و وارد CS نمی‌شود درحالیکه این انتظار بی‌معنی است ← پیشرفت نقض شده

انتظار محدود:

خطوط turn = j و flag[i] = false باعث می‌شود حتی اگر برنامه‌ای بخواهد چندین بار وارد CS شود نتواند زیرا این خطوط اولویت را به برنامه دیگر می‌دهند ← انتظار محدود دارد

TANDIS

ب)

Subject:
Year: Month: Day: ( )

P_i	P_j	
flag[i] = true;	flag[j] = true	1
turn = j;	turn = i;	2
while (flag[j] && turn == j);	while (flag[i] && turn == i);	3
// CS	// CS	4
flag[i] = false	flag[j] = false	5

6

7 انحصار متقابل:

8 از آنجا که turn=i یا turn=j فقط یکی از این دو درست است در نتیجه فقط

9 یکی از فرآیندها می‌تواند وارد CS شود ⇐ انحصار متقابل دارد

10

11 پیشرفت:

12 فرآیند P_i فقط درصورتی وارد CS نمی‌شود که P_j بخواهد کد خود را اجرا کرده

13 باشد و درغیراین صورت (هنگامی که P_j از CS خارج شده باشد یا هنوز به آن

14 نرسیده باشد) وارد CS می‌شود ⇐ پیشرفت دارد

15

16

17

18 انتظار محدود:

19 خط turn = j و flag[i]=false باعث می‌شود حتی اگر فرآیند P_i بخواهد چندین

20 بار وارد CS شود نتواند زیرا با turn=j خودش در while گیر می‌افتد و P_j

21 اجرا می‌شود و یا حتی قبل از آن، با false شدن flag[i] شرط while P_j

22 نقض شده و وارد CS می‌شود ⇐ انتظار محدود برقرار است

23

24

TANDIS

سوال سوم:

```
int multiplication(int op1, int *P_op2){
	int *lock = 0;
	while (compare_and_swap(&lock, 0, 1)){
		*P_op2 = *P_op2 * op1
		lock = 0;
		return *P_op2;

	}
}
bool compare_and_swap (int *value, int old, int new){
	if(*value != old){
		return false;
	}
	*value = new;
	return true;
}
```

شروط انحصار متقابل و پیشرفت را دارد. زیرا تا وقتی فرآیندی lock را ۱ کرده نمی توان وارد حلقه شد پس انحصار متقابل دارد و همچنین حداقل یک فرآیند وارد حلقه می شود (وقتی lock = 0 باشد).

اما ممکن است هیچ وقت نوبت به فرآیندی که درخواست دارد نرسد و شرط انتظار محدود برقرار نیست. برای اینکه این شرط را هم دارا شود، با شماره بندی فرآیند ها و سپس نوبت بندی آنها عدالت را بین آنها برقرار می کنیم.

سوال چهارم:

ب) دستوری به نام compare_and_exchange به صورت اتمیک دو مقدار ۳۲ بیتی را مقایسه می کند و اگر مقدار مقصد و مقدار compared برابر بود مقدار exchange در مقصد ذخیره می شود.