Aseel Khalaf
1160689

19/4/2020
Sunday

Comp431 Operating System

Homework # 3

Question 1

(A)

```
S1;
Count1 = 3;
Count2 = 2;
   Fork L1;
   Fork L2;
   S4;
   S7;
   Fork L9;
   S10;
   Goto J1;
L9: S9;
   Goto J1;
L2: S2;
   S5;
   S8;
   Goto J2;
L3: S3;
   S6;
J1: Joint Count1;
   S11;
J2: Joint Count2;
   S12;
```

(B)

```
S1;
Parbegin;
    Parbegin;
        begin;
            S4;
            S7;
            Parbegin;
                S9;
                S10;
            Parend;
        end;
        begin;
            S3;
            S6;
        end;
    Parend;
    S11;
    begin;
        S2;
        S5;
        S8;
    end;
Parend;
S12;
```

Question 2

(a) Balance for P-A = 600
Balance for P-B = 800

(b) ① A1: load R_1, Balance ⇒ $R_1 = 500$ in P-A
② B1: // R_1 Balance ⇒ $R_1 = 500$ in P-B
③ A2: Add R_1, 100 ⇒ $R_1 = 600$ in P-A
④ B2: Add R_1, 300 ⇒ $R_1 = 900$ in P-B
⑤ A3; Store Balance, R_1 ⇒ Balance = 900 in P-A
⑥ B_3. Store Balance, R_1 ⇒ Balance = 900 in P-B

(c) What is your conclusion regarding the existence of a race in the system?
We have several processes (Two processes P-A & P-B) they are manipulating shared Data, which is the balance (R1) Concurrently, So, the outcome depends on the sequence of the execution for both 2 processes, which leads us to have the amount 900 of the balance, so both processes are racing to reach the critical section.

(d) Semaphores to synchronize A & B so as to avoid races.

```
C = 1;                                   repeat
Wait(C){
critical section { while(C <=0){#do Nothing! just wait}   // Busy waiting
                  C = C - 1; // do operation and turn it to red
                }
Signal(C){
reminder section { C = C + 1;   // when the process is done we have to turn it into Green
                }
Forever
```

C=1 Green
C=0 Red
check critical section

```
Mutex : C=1;
 Repeat
  Wait (Mutex);
     Critical section;
   Signal(Mutex);
     Remainder section
  Forever
```

So, we solve the race problem by let the processes to work atomically in the critical system to avoid race

Question 3

(a) Data structures:

* Available array :- which indicates the number of available (m) resources of each type, so Ex: available[m] = 5 which means there is available 5 of type m currently.

* Needs array :- which is 2d array n x m, indicates the current request of each process

  Needs[i,j] = r then process P is requesting r instances of type Need[i,j].

* Allocation array :- (n x m array) indicates the number of resources of each type which is currently allocated to the processes. ex: allocation[i,j] = 4

  for a specific processes for specific resourse it have 4 allocated currently.

Algorithm:-

Let W (available resources) and F (Finish process) be Vectors of length m & n // because we have multi resources not 1

[1] Initialize:-

W := available

F[i] = Finish = False for i = 1, 2, 3 ..., n

[2] Search for an i such that (F[i] = false && Need[i] ≤ W)

IF (false (no item meets the condition),

Skip toward number 4

[3] W := W + allocation i // if the condition true, do the process

F[i] = true // set it to, true (process done)

⇒ Repeat step 2

[4] if F[i] = true for all the process then system safe

else

Not safe // if there is one at least is false

(b) 5 processes $P_0 \rightarrow P_4$

3 Resources — A ↓ 7 instances, B ↓ 2 instances, C ↓ 6 instances

	A	B	C
Available	7	2	6

	Allocation A B C	Max A B C	Request A B C
P_0	0 1 0	0 1 0	0 0 0
P_1	2 0 0	4 0 2	2 0 2
P_2	3 0 3	3 0 3	0 0 0
P_3	2 1 1	3 1 1	1 0 0
P_4	0 0 2	0 0 2	0 0 2

Available A B C
0 0 0 + 0 1 0
0 1 0 + 3 0 3
3 1 3 + 2 1 1
5 2 4 + 2 0 0
7 2 4 + 0 0 2
7 2 6

① Needs for $P_0 = (000) \leq$ available (000) So, aVailable = aVailable + allocation P_0
= 0 0 0 + 0 1 0
= **0 1 0**

② " " $P_1 = 202 \not\leq (010)$ NO so, skip to P_2

③ " " $P_2 = (000) \leq 010$ ✓ so, aVailable = 010 + 303
= 313

④ " " $P_3 = (100) \leq 313$ ✓ " aVailable = 313 + 211
= 524

⑤ Now, we can execute P_1 ⇒ aVailable = 524 + 200
= 724

⑥ $P_4 = 002 \leq 724$, aVailable = 724 + 002
= **726**

Sequence : P_0, P_2, P_3, P_1, P_4

(i) Yes, the system is safe, No deadlocks

* Finish[i] = true for all processes

(ii) if P_2 Refurst additional instance of C.

	Allocation A B C	Refurst A B C	available A B C
P_0	0 1 0	0 0 0	0 0 0 + 0 1 0
P_1	2 0 0	2 0 2	0 1 0
P_2	3 0 1	0 0 1	Deadlock!
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Deadlock! because: $P_0 \Rightarrow 000 \leq 000$ then available = 000 + 010 = 010

$P_1 \Rightarrow 202 > 010$
$P_2 \Rightarrow 001 > 010$
$P_3 \Rightarrow 100 > 010$
$P_4 \Rightarrow 002 > 010$

So, the deadlock consist of processes P_1, P_2, P_3, P_4.