Chapter 6

Q 6.1. Satisfies ME by providing toggled access; Progress ppty not satisfied.

Q 6.2. Can mutually block indefinitely; again, progress ppty not satisfied;

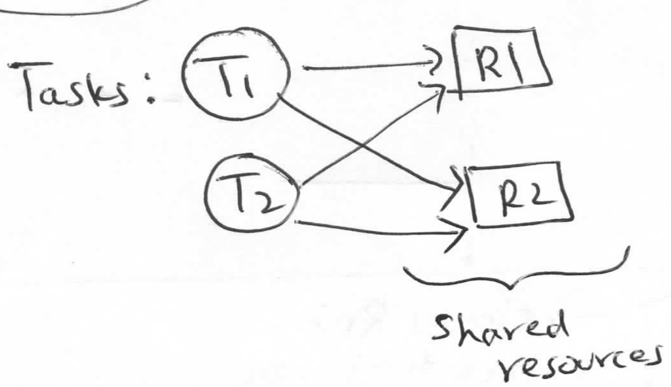
Q 6.3. Does not satisfy ME;

Q.6.4. All 3 ppt'ies satisfied;

Q6.6. Bounded waiting time is not satisfied; leads to indefinite waiting wasting CPU cycles;

Q6.9 (d)

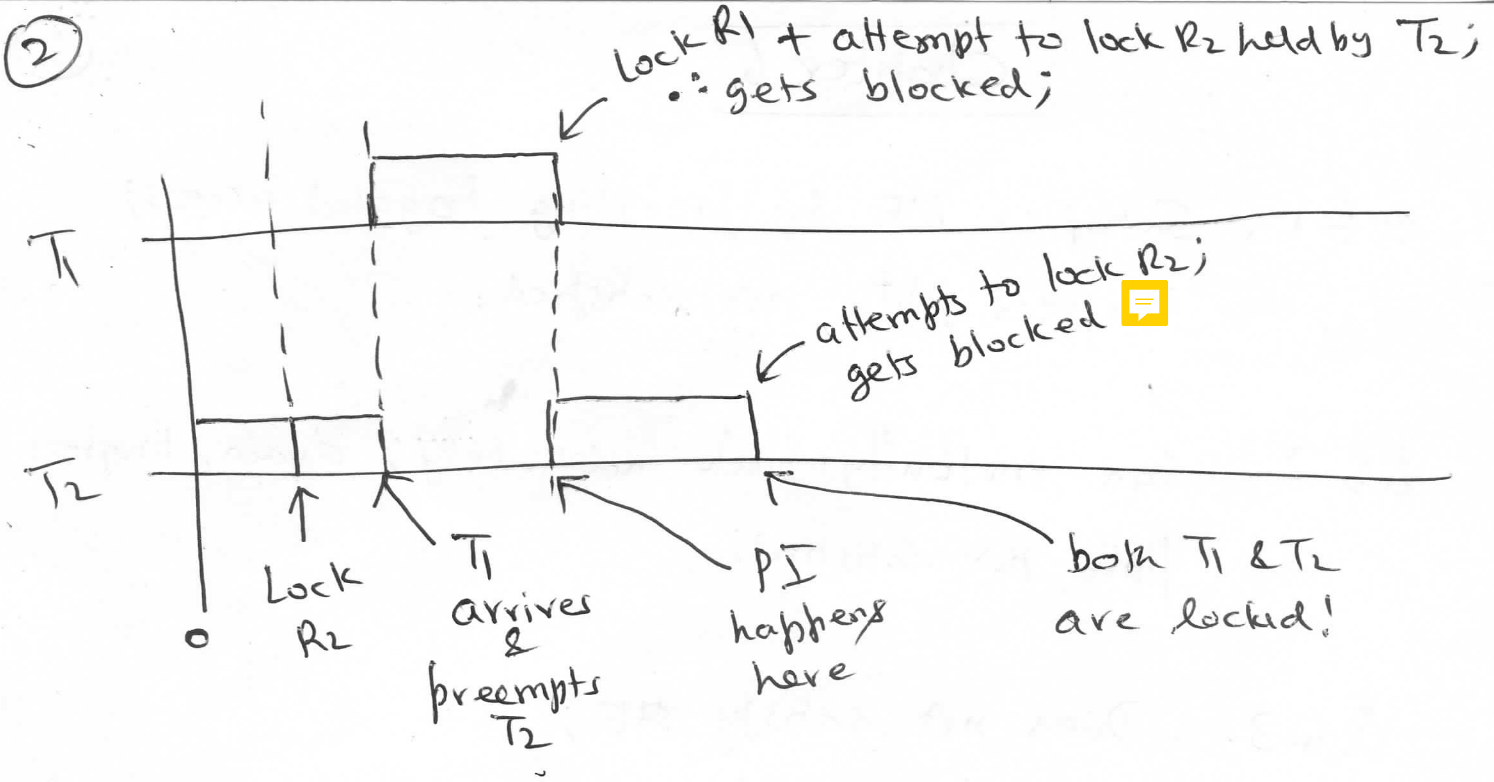
Q6.10 (deadlock scenario)



Initial condition

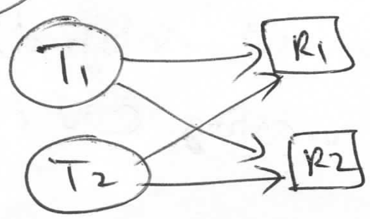
- priority(T_1) > priority(T_2)
- T_2 starts running first as T_1 is not ready yet;

(2)

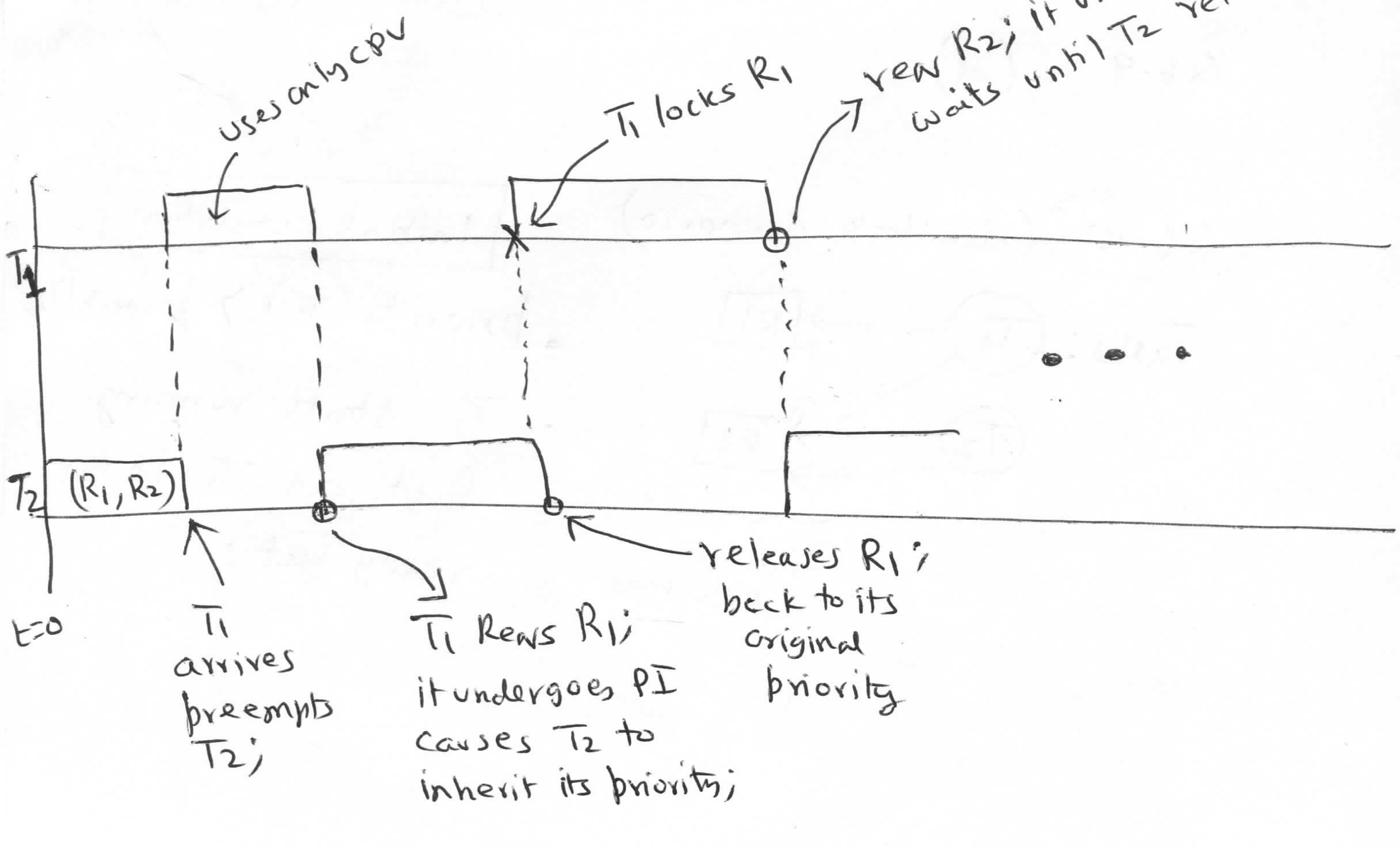


Q 6.11 pri(T2) < pri(T1)

T2 ready @ t=0 holds both resources R1 & R2;



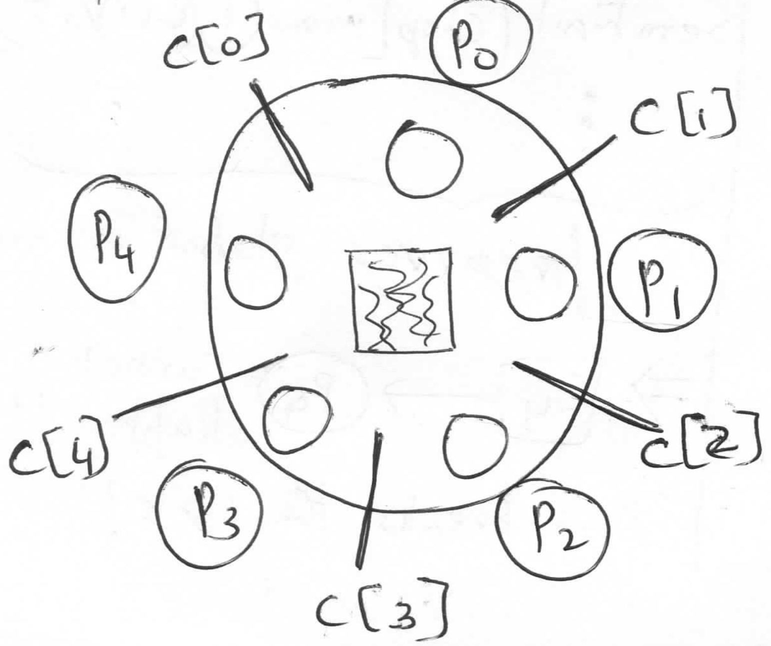
Chain Blocking



Slide on Dining Philosopher's problem

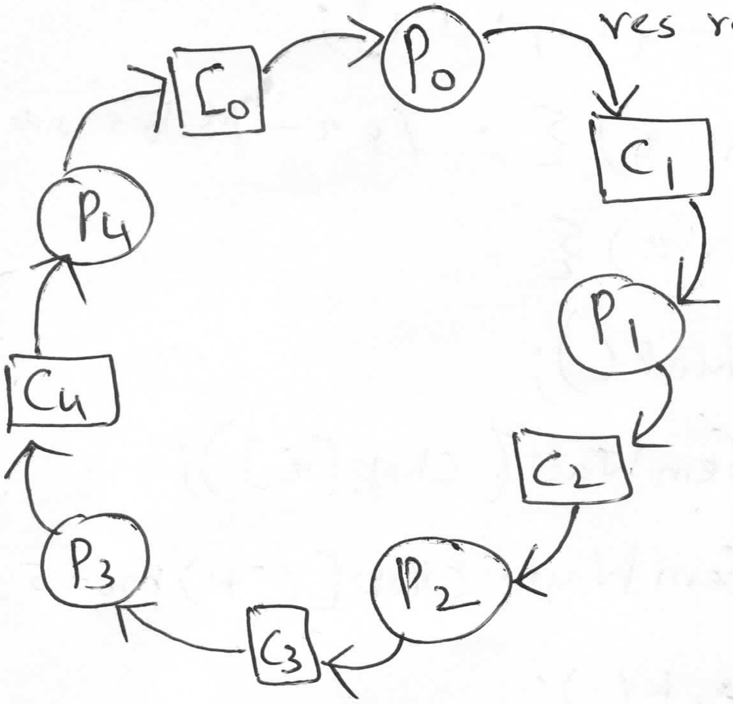
Attempt-1

```
Sem [] chop = {1, 1, 1, 1, 1};
void philo (int i) {    /* i - philo's index */
    while (true) {
        think();
        SemWait( Chop[i]);
        SemWait( Chop[(i+1) mod 5]);
        eat();
        SemSignal (Chop[ (i+1) mod 5]);
        SemSignal (Chop [i]);
    }
    return 0;
}
```



4

Philo i → first grabs chop on his/her right;
→ then onto his/her left;



a cycle! & hence a deadlock!

how to avoid this deadlock?

P	R	L
0	0	1
1	1	2
2	2	3
3	3	4
4	4	0

Attempt-1 rule

Correct Solution

⋮

```
think();
SemWait(chop[min(i,(i+1)%5)]);
SemWait(chop[max(i,(i+1)%5)]);
```

⋮

preserves global ordering

$\Rightarrow \boxed{C_4} \longrightarrow P_4$ cannot happen!!

This breaks the cycle!

Deadlock Avoidance – Banker's Algo.

- Scan from $P_0 \rightarrow P_1 \rightarrow \cdots \rightarrow P_4$ to identify the safe seqvence;

- Need [i] = Max [i] – Allocation [i]

  completed matrix shown in your slide.

- Then we need to check if processes P_0 to P_4 can be run by allocating the reqvested (max) number of resources. For this, we need to generate Available [i] & check against the Need [i]; For example:

P_0 : $\left.\begin{array}{l} A \rightarrow 10-7=3 \\ B \rightarrow 5-2=3 \\ C \rightarrow 7-5=2 \end{array}\right\}$ Available [0] = (3 3 2) which cannot be satisfied with a Need [0] = (7 4 3).

Hence, we proceed to check P_1;

Available [0] = (3 3 2) > Need [1]

$\Rightarrow$ P_1 can be run;

after P_1 completes it releases its resources } Available [1] = (3 3 2) + (1 2 2) = (4 5 4)

Continue as above & generate the solution safe seq: (1, 3, 4, 0, 2)*.