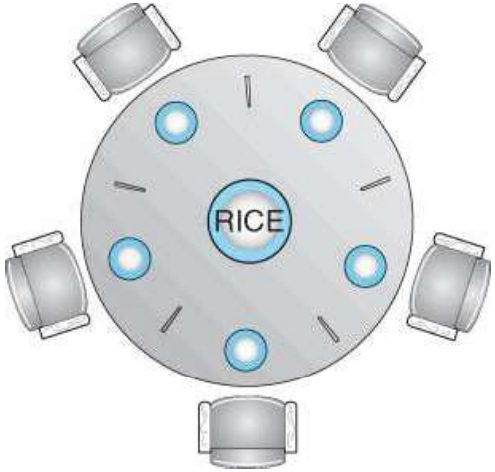
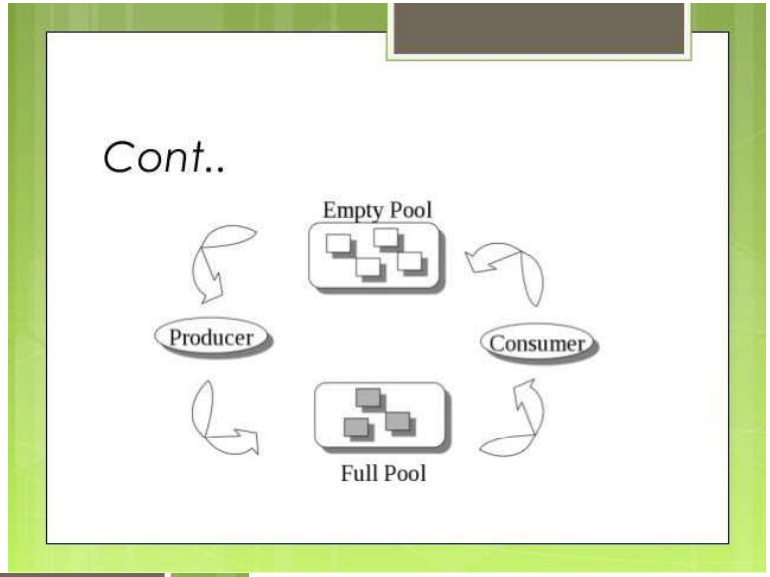
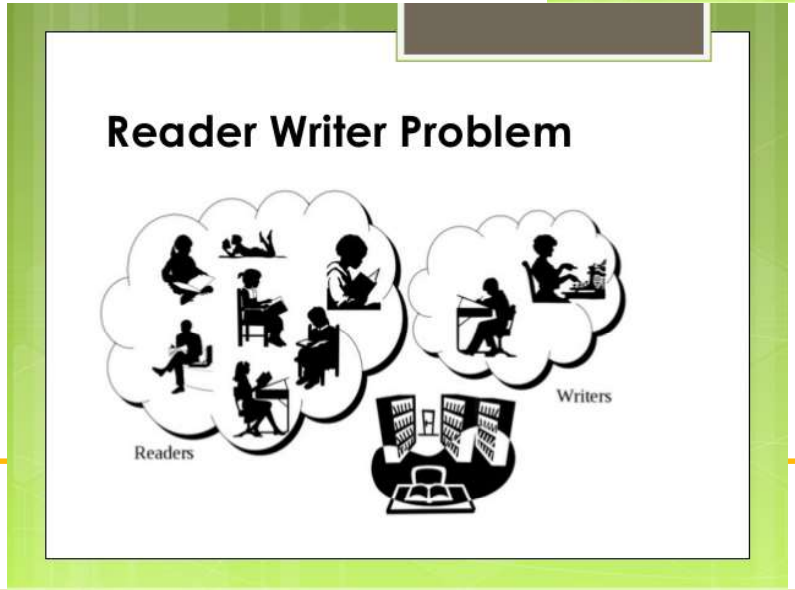
Some Classical CSP Problems

- Dining Philosophers Problem ; Reader Writer Problem
- Producer Consumer Problem

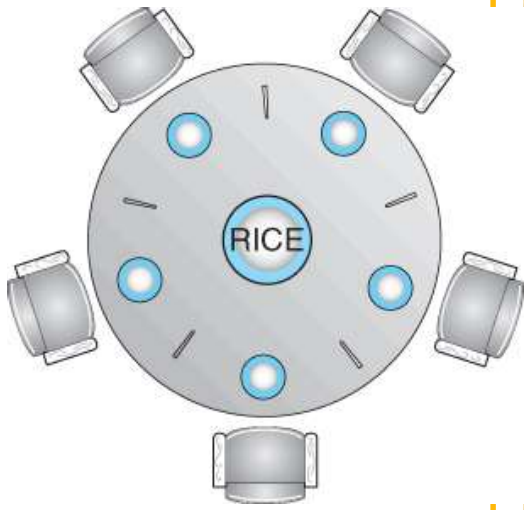






DINING PHILOSOPHERS PROBLEM

- Classical CSP- allocation of limited resources a (chopsticks) to philosophers in deadlock-free and starvation-free manne
- 5 philosophers sitting around a table ; five chopsticks evenly distributed and an endless bowl of rice in the center;
- exactly one chopstick between each pair of dining philosophers.
- philosophers spend their lives alternating between two activities: eating and thinking
- for a philosopher to eat, it must first acquire 2 chopsticks - one from left and one from right.
- philosopher thinks, puts down both chopsticks in their original locations.



- five semaphores **(chopsticks[5]),** and to have each hungry philosopher first wait on their left chopstick(chopsticks[i]);
- wait on their **right chopstick (chopsticks[(i + 1) % 5])**
- If all **5 philosophers get hungry at the same time**;
- each starts by picking up their left chopstick.;
- **look for their right chopstick,** but because it is unavailable, **they wait for it, forever; philosophers starve due to the resulting deadlock.**

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   // eat
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   // think
      . . .
}while (TRUE);
```

- Only **allow four philosophers to dine at the same time**. (Limited simultaneous processes.)
- Allow philosophers to pick up chopsticks only when both are available, in a critical section. (All or nothing allocation of critical resources.)
- Use an asymmetric solution, in which odd philosophers pick up their left chopstick first and even philosophers pick up their right chopstick first.
- deadlock-free solution to the dining philosophers problem does not necessarily guarantee a starvation-free one
- (Will this solution always work? What if there are an even number of philosophers?)

```
#include<stdio.h>        #include<semaphore.h>
#include<pthread.h>     #define N 5
#define THINKING 0    #define HUNGRY 1
#define EATING 2        #define LEFT (ph_num+4)%N
#define RIGHT (ph_num+1)%N
 sem_t mutex; sem_t S[N];
 void * philospher(void *num);   void take_fork(int);
void put_fork(int);         void test(int);
int state[N];
int phil_num[N]={0,1,2,3,4};
```

```
int main()
{
    int i;  pthread_t thread_id[N];

    sem_init(&mutex,0,1);
    for(i=0;i<N;i++)
        sem_init(&S[i],0,0);
    for(i=0;i<N;i++)
    {  pthread_create(&thread_id[i],NULL,philospher,&phil_num[i]);
        printf("Philosopher %d is thinking\n",i+1);
    }
    for(i=0;i<N;i++)
    pthread_join(thread_id[i],NULL);
}
```

```
void *philospher(void *num)
   {    while(1)    {
        int *i = num;          sleep(1);
        take_fork(*i);         sleep(0);
        put_fork(*i);
    }}
   void take_fork(int ph_num)
{
    sem_wait(&mutex);
    state[ph_num] = HUNGRY;
    printf("Philosopher %d is Hungry\n",ph_num+1);
    test(ph_num);
    sem_post(&mutex);
    sem_wait(&S[ph_num]);
   sleep(1);
}
```

```
void test(int ph_num)
{
    if (state[ph_num] == HUNGRY && state[LEFT] != EATING
    && state[RIGHT] != EATING)
    {  state[ph_num] = EATING;
        sleep(2);
printf("Philosopher %d takes fork %d and
    %d\n",ph_num+1,LEFT+1,ph_num+1);
        printf("Philosopher %d is Eating\n",ph_num+1);
        sem_post(&S[ph_num]);
    }
}
```

```
void put_fork(int ph_num)
{
    sem_wait(&mutex);
    state[ph_num] = THINKING;
    printf("Philosopher %d putting fork %d and %d
    down\n",ph_num+1,LEFT+1,ph_num+1);
    printf("Philosopher %d is thinking\n",ph_num+1);
    test(LEFT);
    test(RIGHT);
    sem_post(&mutex);
}
```