Using Monitor	**Equivalent Mapping**

Monitor DiningPhilosophers
{
enum { THINKING; HUNGRY, EATING) state [5] ;
condition self [5];
void `grab_fork` (int i) {
state[i] = HUNGRY;
test(i);
if (state[i] != EATING)
self[i].wait;
}
 	
void `put_fork` (int i) {
state[i] = THINKING;
// test left and right
neighbors
test((i + 4) % 5);
test((i + 1) % 5);
}
	`enum st{THINKING, EATING, HUNGRY};`
	`enum st *state;`
	
	void grab_fork(int num)
	{
	pthread_mutex_lock(&mt);
	printf("\nPhilospher[%d] is hungry\n",num);
	//sleep(rand()%5);
	state[num] = HUNGRY;
	test(num);
	pthread_mutex_unlock(&mt);
	sem_wait(&cs[num]);
	}
	
	void put_fork(int num)
	{
	pthread_mutex_lock(&mt);
	state[num] = THINKING;
	test(num);
	test((num+1)%N);
	pthread_mutex_unlock(&mt);
	}

```
void test (int i) {
 if ((state[(i + 4) % 5] != EATING)
&&
(state[i] == HUNGRY) &&
(state[(i + 1) % 5] != EATING) ) {
state[i] = EATING ;
self[i].signal () ;
}
}
 initialization_code() {
    for (int i = 0; i < 5; i++)
      state[i] = THINKING;
 }
}
```

```
void test(int i)
{
if(state[i]==HUNGRY && state[(i+4)%N] != EATING &&
state[(i+1)%N] != EATING)
{
  printf("\nPhilospher[%d] is eating\n",i);
  sleep(rand()%4);
 state[i] = EATING;
  sem_post(&cs[i]);
 }
}
```

```
sem_t *cs;
pthread_mutex_t mt;

void* philospher(void*);
void grab_fork(int);
void put_fork(int);
void test(int);
void think(int);
void eat(int);

int main()
{
      pthread_t *phil;
      int i,err;
      int *index;

      printf("\nEnter no. of philosphers(NOTE: It is
assumed that no. of forks = no. of philosphers):\n");
      scanf("%d",&N);

      //MUTEX INITIALIZATION
      pthread_mutex_init(&mt, NULL);

      //DYNAMIC MEMORY ALLOCATION
      phil = (pthread_t*) malloc(N * sizeof(pthread_t));
      index = (int*) malloc(N * sizeof(int));
      cs = (sem_t*) malloc(N * sizeof(sem_t));
      state = (enum st*) malloc(N * sizeof(enum st));
```

Philosopher Structure

```
do {
   wait (chopstick[i] );
   wait (chopStick[ (i + 1) % 5] );

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

```
      //INITIALIZATION
      for(i=0;i<N;i++)
      {
       state[i] = THINKING;
      sem_init(&cs[i],0,0); //binary sem are initialised
by 0
      }

      //THREADS CREATION
      for(i=0;i<N;i++)
      {
            index[i]=i;
            err
pthread_create(&phil[i],NULL,philospher,(void*)&index[i]);
            if(err!=0)
            {
                  printf("\nError in thread creation!!!");
                  exit(0);
            }
      }

      //THREADS JOINING
      for(i=0;i<N;i++){
            err = pthread_join(phil[i],NULL);
            if(err!=0)
            {
                  printf("\nError in thread joining!!!");
                  exit(0);
            }
      }

return 0;
}

void *philospher(void *arg)
{
      int i = *(int*)arg;
      while(1)
      {
            printf("\nPhilospher[%d] is thinking\n",i);
            //sleep(rand()%3);
            grab_fork(i);
            put_fork(i);
      }
}
```
