Algorithm

```
#define N               5               /* number of philosophers */
#define LEFT            (i+N−1)%N       /* number of i's left neighbor */
#define RIGHT           (i+1)%N         /* number of i's right neighbor */
#define THINKING        0               /* philosopher is thinking */
#define HUNGRY          1               /* philosopher is trying to get forks */
#define EATING          2               /* philosopher is eating */
typedef int semaphore;                  /* semaphores are a special kind of int */
int state[N];                           /* array to keep track of everyone's state */
semaphore mutex = 1;                    /* mutual exclusion for critical regions */
semaphore s[N];                         /* one semaphore per philosopher */

void philosopher(int i)                 /* i: philosopher number, from 0 to N−1 */
{
    while (TRUE) {                      /* repeat forever */
        think( );                       /* philosopher is thinking */
        take_forks(i);                  /* acquire two forks or block */
        eat( );                         /* yum-yum, spaghetti */
        put_forks(i);                   /* put both forks back on table */
    }
}


void take_forks(int i)                  /* i: philosopher number, from 0 to N−1 */
{
    down(&mutex);                       /* enter critical region */
    state[i] = HUNGRY;                  /* record fact that philosopher i is hungry */
    test(i);                            /* try to acquire 2 forks */
    up(&mutex);                         /* exit critical region */
    down(&s[i]);                        /* block if forks were not acquired */
}

void put_forks(i)                       /* i: philosopher number, from 0 to N−1 */
{
    down(&mutex);                       /* enter critical region */
    state[i] = THINKING;                /* philosopher has finished eating */
    test(LEFT);                         /* see if left neighbor can now eat */
    test(RIGHT);                        /* see if right neighbor can now eat */
    up(&mutex);                         /* exit critical region */
}

void test(i)                            /* i: philosopher number, from 0 to N−1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

Program

```
#include <stdio.h>
#include <semaphore.h>
#include <unistd.h>
#include <pthread.h>

#define N 5
#define LEFT (i+N-1)%N
#define RIGHT (i+1)% N
#define THINKING 0
#define HUNGRY 1
#define EATING 2

int i=1;
int state[N];
sem_t mutex;
sem_t s[N];
void test(int);
void take_forks(int);
void put_forks(int);

void * philosopher(void *i)
{
        int *p = (int *) i;
        while(1)
        {
                printf("\n philosopher %d is thinking",*p);
                sleep(5);
                take_forks(*p);
                printf("\n philosopher %d is eating",*p);
                sleep(5);
                put_forks(*p);
}
}

void take_forks(int i)
{
        sem_wait(&mutex);
        state[i] = HUNGRY;
        printf("\n philosopher %d is in hungry state",i);
        test(i);
        sem_post(&mutex);
        sem_wait(&s[i]);
```

```
}

void put_forks(int i)
{
        sem_wait(&mutex);
        state[i] = THINKING;
        test(LEFT);
        test(RIGHT);
        sem_post(&mutex);
}

void test(int i)
{
        if(state[i]== HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)
        {
                state[i]= EATING;
                sem_post(&s[i]);
        }
}

int main()
{
        pthread_attr_t *attr= NULL;
        pthread_t p_tid1,p_tid2,p_tid3,p_tid4,p_tid5;
        sem_init(&mutex,0,1);
        int p[5]={0,1,2,3,4};
        pthread_create(&p_tid1,attr,philosopher,(void *) &p[0]);
        pthread_create(&p_tid2,attr,philosopher,(void *) &p[1]);
        pthread_create(&p_tid3,attr,philosopher,(void *) &p[2]);
        pthread_create(&p_tid4,attr,philosopher,(void *) &p[3]);
        pthread_create(&p_tid5,attr,philosopher,(void *) &p[4]);
        pthread_join(p_tid1,NULL);
        pthread_join(p_tid2,NULL);
        pthread_join(p_tid3,NULL);
        pthread_join(p_tid4,NULL);
        pthread_join(p_tid5,NULL);
        return 0;
}
```