

Tutorial Week 2 (for discussion Week 3)

- Q1. Following is the textbook's solution to the Dining Philosophers problem.

- (i) What does "test" do?
- (ii) Explain why this solution is superior to a single mutex.
-

```
#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]);
    }
}

....cont/
```

Q2. Following is the textbook's solution to the readers & writers problem.

(i) Assume "read_data_base()" takes 3 seconds regardless of number of readers, and "write_data_base()" takes 4 seconds.

(ii) Show which calls are active, and which are blocked, if reads are requested at time 0,2,3,5,6,8 seconds, and writes are requested at time 4, 7 seconds.

```
typedef int semaphore;          /* use your imagination */
semaphore mutex = 1;          /* controls access to 'rc' */
semaphore db = 1;             /* controls access to the database */
int rc = 0;                    /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {             /* repeat forever */
        down(&mutex);          /* get exclusive access to 'rc' */
        rc = rc + 1;           /* one reader more now */
        if (rc == 1) down(&db); /* if this is the first reader ... */
        up(&mutex);            /* release exclusive access to 'rc' */
        read_data_base();      /* access the data */
        down(&mutex);          /* get exclusive access to 'rc' */
        rc = rc - 1;           /* one reader fewer now */
        if (rc == 0) up(&db);  /* if this is the last reader ... */
        up(&mutex);            /* release exclusive access to 'rc' */
        use_data_read();       /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {             /* repeat forever */
        think_up_data();       /* noncritical region */
        down(&db);             /* get exclusive access */
        write_data_base();     /* update the data */
        up(&db);               /* release exclusive access */
    }
}
```