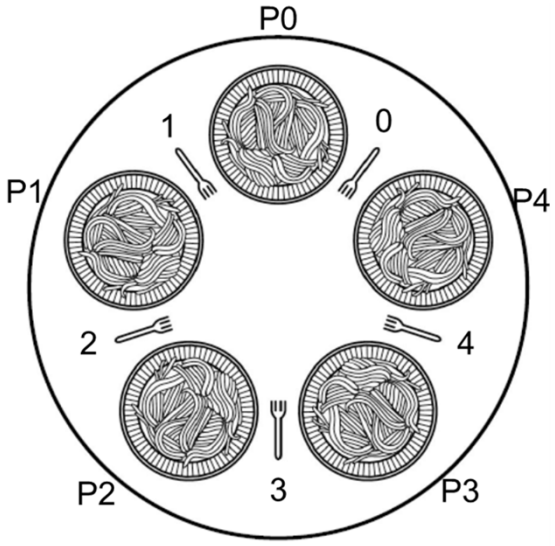
作業系統
Operating Systems

靜宜大學資訊傳播工程學系
劉國有 助理教授
kyliu@pu.edu.tw

The Dinning Philosophers Problem



The Dinning Philosophers Problem – *Solution 1*

```
#define N 5                                  /* number of philosophers */

void philosopher(int i)                      /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                            /* philosopher is thinking */
        take_fork(i);                        /* take left fork */
        take_fork((i+1) % N);                /* take right fork; % is modulo operator */
        eat( );                              /* yum-yum, spaghetti */
        put_fork(i);                         /* put left fork back on the table */
        put_fork((i+1) % N);                 /* put right fork back on the table */
    }
}
```

- Deadlock: all five philosophers take their left forks simultaneously.
- 解法：先拿左邊，如右邊有人佔住，則放下，等待一段時間再拿(會有starvation)

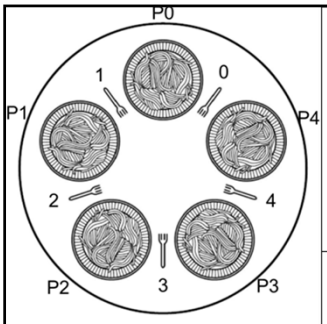
The Dinning Philosophers Problem – *Solution 1*

- Starvation: all the programs continue to run indefinitely but fail to make any progress.[同時拿起左邊，一起放下左邊，等待一段時間，再同時拿起左邊。解決：等待的時間利用亂數產生]
- No deadlock and no starvation solution: to protect the five statements following the call to think() by a binary semaphore. [performance bug: only one philosopher can be eating at any instant.]

The Dinning Philosophers Problem – *Solution 2*

```
#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]);
    }
}
```

The Readers and Writers

- It is acceptable to have multiple processes reading the database at the same time
- If one process is updating (writing) the database, no other process may have access to the database.

```
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 */
        }
}
```