Reader-Writers problem

- The resource is a file shared by multiple reader and writer processes
- The synchronization constraints are:
  - readers should be able to concurrently access the file
  - only one writer at a time can access the file
  - readers and writers exclude each others
- Variants:
  - reader's priority: arriving readers have priority over waiting writers
  - writer's priority: writers have priority over waiting readers

Simple Readers-Writers solution

- The following scheme is very simple but …..
- … does not allow concurrent reader access

Procedure reader

P(mutex)

<*read file*>

V(mutex)

Procedure writer

P(mutex)

<*write file*>

V(mutex)

Readers-Writers solution with concurrent reader access

Procedure reader

```
P(reader_mutex)
if readers = 0 then
    readers = readers + 1
    P(writer_mutex)
else
    readers = readers + 1
V(reader_mutex)
```

<read file>

```
P(reader_mutex)
readers = readers - 1
 if readers == 0 then V(writer_mutex)
V(reader_mutex)
```

Procedure writer

```
P(writer_mutex)

<write file>

V(writer_mutex)
```

This solution is **NOT ALWAYS** with reader's priority. WHY?

Readers-Writers with reader's priority

Procedure reader

```
P(reader_mutex)
if readers = 0 then
    readers = readers + 1
    P(writer_mutex)
else
    readers = readers + 1
V(reader_mutex)

<read file>

P(reader_mutex)
readers = readers - 1
 if readers == 0 then V(writer_mutex)
V(reader_mutex)
```

Procedure writer

```
P(sr_mutex)
P(writer_mutex)

<write file>

V(writer_mutex)
V(sr_mutex)
```

Readers-Writers with reader's priority

Procedure reader

```
P(reader_mutex)
if readers = 0 then
    P(writer_mutex)
readers = readers + 1
V(reader_mutex)

<read file>

P(reader_mutex)
readers = readers - 1
if readers = 0 then V(writer_mutex)
V(reader_mutex)
```

What if this call is omitted?

Procedure writer

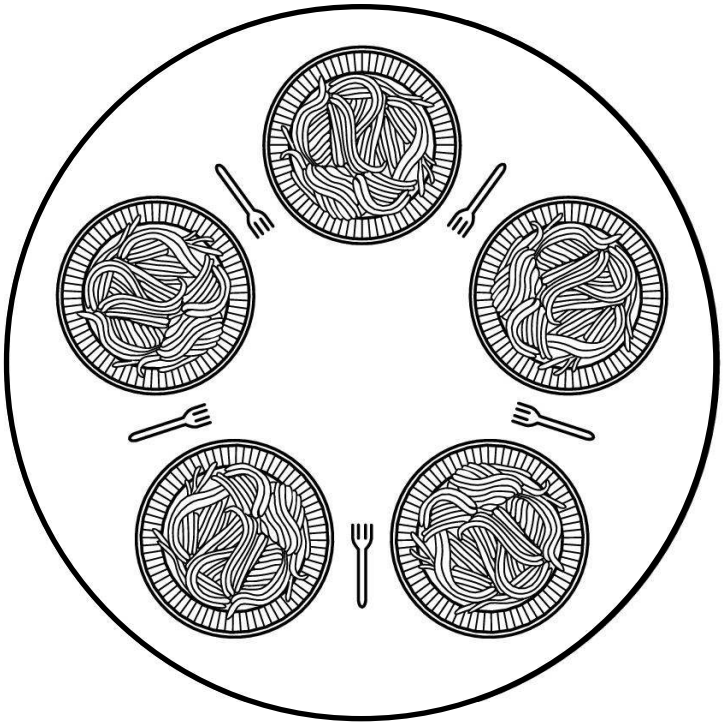
```
P(sr_mutex)
P(writer_mutex)

<write file>

V(writer_mutex)
V(sr_mutex)
```

Dining Philosophers Problem

- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- Possible deadlock?
- How to prevent deadlock?



Does it solve the Dining Philosophers Problem?

```
#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 */
    }
}
```

NOT a solution to the dining philosophers problem

Dining Philosophers Solution

```
#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 */
    }
}
```

How to implement take_forks() and put_forks()?

Dining Philosophers Solution

```
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]);
      }
}
```

Dining Philosophers Solution

```
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]);
      }
}
```

Dining Philosophers Solution

```
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]);
    }
}
```

Dining Philosophers Solution

```
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]);
      }
}
```