Operating Systems and Concurrency

Lecture 11: Concurrency

G52OSC/COMP2007

Geert De Maere
(Isaac Triguero)
{Geert.DeMaere,Isaac.Triguero}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2018

Goals
Today

- Module feedback:
  - Pace of the lectures (70% about right)
  - Incentives to do the labs
- Parallel dining philosophers
- Readers/writers problem

The Dining Philosophers Problem

Solutions 2: Global Mutex/Semaphore

Question after the last lecture:

"Can I initialise the value of the `eating` *semaphore to 2 to create more parallelism"*

The Dining Philosophers Problem
Solutions 2: Global Mutex/Semaphore

```
sem_t eating;

void * philosopher(void * id)
{
  int i = (int) id;
  int left = (i + N - 1) % N;
  int right = i % N;
  while(1)
  {
    printf("%d is thinking\n", i);
    printf("%d is hungry\n", i);
    sem_wait(&eating);       /**** semaphore ****/
    sem_wait(&forks[left]);
    sem_wait(&forks[right]);
    printf("%d is eating\n", i);
    sem_post(&forks[left]);
    sem_post(&forks[right]);
    sem_post(&eating);       /**** semaphore ****/
  }
}
```

The Dining Philosophers Problem

Solutions 2: Illustration

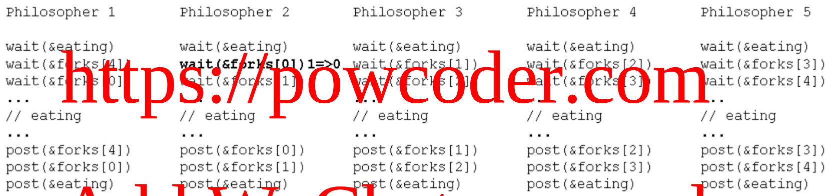
```
Philosopher 1      Philosopher 2      Philosopher 3      Philosopher 4      Philosopher 5

wait(&eating)      wait(&eating)      wait(&eating)      wait(&eating)      wait(&eating)
wait(&forks[4])    wait(&forks[0])    wait(&forks[1])    wait(&forks[2])    wait(&forks[3])
wait(&forks[0])    wait(&forks[1])    wait(&forks[2])    wait(&forks[3])    wait(&forks[4])
...                ...                ...                ...                ...
// eating          // eating          // eating          // eating          // eating
...                ...                ...                ...                ...
post(&forks[4])    post(&forks[0])    post(&forks[1])    post(&forks[2])    post(&forks[3])
post(&forks[0])    post(&forks[1])    post(&forks[2])    post(&forks[3])    post(&forks[4])
post(&eating)      post(&eating)      post(&eating)      post(&eating)      post(&eating)
```

The Dining Philosophers Problem

Solutions 2: Illustration

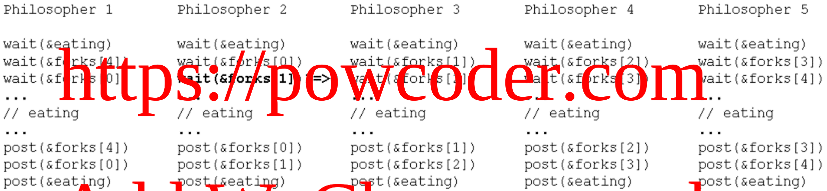
```
Philosopher 1       Philosopher 2       Philosopher 3       Philosopher 4       Philosopher 5

wait(&eating)       wait(&eating)2=>1   wait(&eating)       wait(&eating)       wait(&eating)
wait(&forks[4])     wait(&forks[0])     wait(&forks[1])     wait(&forks[2])     wait(&forks[3])
wait(&forks[0])     wait(&forks[1])     wait(&forks[2])     wait(&forks[3])     wait(&forks[4])
...                 ...                 ...                 ...                 ...
// eating           // eating           // eating           // eating           // eating
...                 ...                 ...                 ...                 ...
post(&forks[4])     post(&forks[0])     post(&forks[1])     post(&forks[2])     post(&forks[3])
post(&forks[0])     post(&forks[1])     post(&forks[2])     post(&forks[3])     post(&forks[4])
post(&eating)       post(&eating)       post(&eating)       post(&eating)       post(&eating)
```

The Dining Philosophers Problem
Solutions 2: Illustration



```
Philosopher 1        Philosopher 2        Philosopher 3        Philosopher 4        Philosopher 5

wait(&eating)        wait(&eating)        wait(&eating)        wait(&eating)        wait(&eating)
wait(&forks[4])      wait(&forks[0])1=>0  wait(&forks[1])      wait(&forks[2])      wait(&forks[3])
wait(&forks[0])      wait(&forks[1])      wait(&forks[2])      wait(&forks[3])      wait(&forks[4])
...                  ...                  ...                  ...                  ...
// eating            // eating            // eating            // eating            // eating
...                  ...                  ...                  ...                  ...
post(&forks[4])      post(&forks[0])      post(&forks[1])      post(&forks[2])      post(&forks[3])
post(&forks[0])      post(&forks[1])      post(&forks[2])      post(&forks[3])      post(&forks[4])
post(&eating)        post(&eating)        post(&eating)        post(&eating)        post(&eating)
```

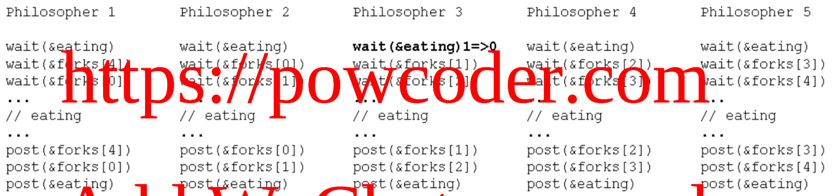
The Dining Philosophers Problem
Solutions 2: Illustration



```
Philosopher 1       Philosopher 2          Philosopher 3       Philosopher 4       Philosopher 5

wait(&eating)       wait(&eating)          wait(&eating)       wait(&eating)       wait(&eating)
wait(&forks[4])     wait(&forks[0])        wait(&forks[1])     wait(&forks[2])     wait(&forks[3])
wait(&forks[0])     wait(&forks[1]) =>     wait(&forks[2])     wait(&forks[3])     wait(&forks[4])
...                 ...                    ...                 ...                 ...
// eating           // eating              // eating           // eating           // eating
...                 ...                    ...                 ...                 ...
post(&forks[4])     post(&forks[0])        post(&forks[1])     post(&forks[2])     post(&forks[3])
post(&forks[0])     post(&forks[1])        post(&forks[2])     post(&forks[3])     post(&forks[4])
post(&eating)       post(&eating)          post(&eating)       post(&eating)       post(&eating)
```

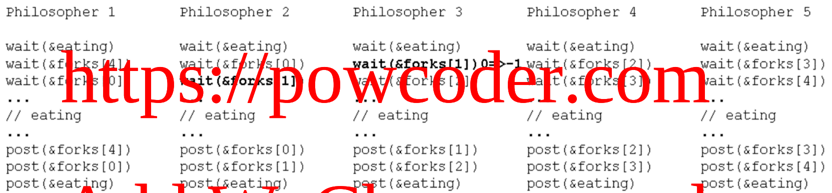
The Dining Philosophers Problem

Solutions 2: Illustration

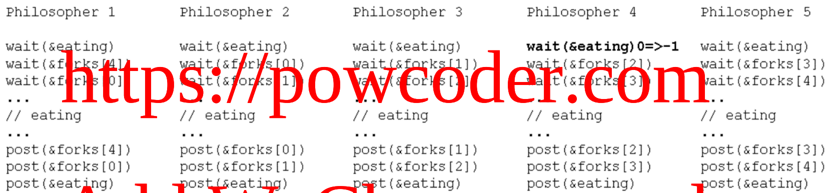


```
Philosopher 1       Philosopher 2       Philosopher 3       Philosopher 4       Philosopher 5

wait(&eating)       wait(&eating)       wait(&eating)1=>0   wait(&eating)       wait(&eating)
wait(&forks[4])     wait(&forks[0])     wait(&forks[1])     wait(&forks[2])     wait(&forks[3])
wait(&forks[0])     wait(&forks[1])     wait(&forks[2])     wait(&forks[3])     wait(&forks[4])
...                 ...                 ...                 ...                 ...
// eating           // eating           // eating           // eating           // eating
...                 ...                 ...                 ...                 ...
post(&forks[4])     post(&forks[0])     post(&forks[1])     post(&forks[2])     post(&forks[3])
post(&forks[0])     post(&forks[1])     post(&forks[2])     post(&forks[3])     post(&forks[4])
post(&eating)       post(&eating)       post(&eating)       post(&eating)       post(&eating)
```

The Dining Philosophers Problem
Solutions 2: Illustration



```
Philosopher 1        Philosopher 2        Philosopher 3           Philosopher 4        Philosopher 5

wait(&eating)        wait(&eating)        wait(&eating)           wait(&eating)        wait(&eating)
wait(&forks[4])      wait(&forks[0])      wait(&forks[1])0=>-1    wait(&forks[2])      wait(&forks[3])
wait(&forks[0])      wait(&forks[1])      wait(&forks[2])         wait(&forks[3])      wait(&forks[4])
...                  ...                  ...                     ...                  ...
// eating            // eating            // eating               // eating            // eating
...                  ...                  ...                     ...                  ...
post(&forks[4])      post(&forks[0])      post(&forks[1])         post(&forks[2])      post(&forks[3])
post(&forks[0])      post(&forks[1])      post(&forks[2])         post(&forks[3])      post(&forks[4])
post(&eating)        post(&eating)        post(&eating)           post(&eating)        post(&eating)
```



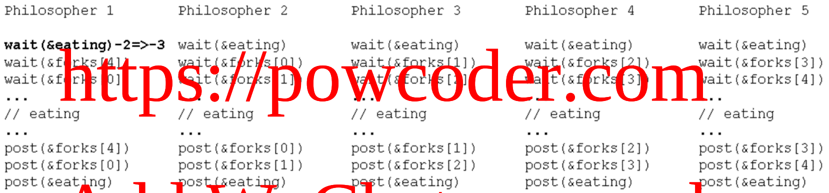
```
Philosopher 1      Philosopher 2      Philosopher 3      Philosopher 4          Philosopher 5

wait(&eating)      wait(&eating)      wait(&eating)      wait(&eating)0=>-1     wait(&eating)
wait(&forks[4])    wait(&forks[0])    wait(&forks[1])    wait(&forks[2])        wait(&forks[3])
wait(&forks[0])    wait(&forks[1])    wait(&forks[2])    wait(&forks[3])        wait(&forks[4])
...                ...                ...                ...                    ...
// eating          // eating          // eating          // eating              // eating
...                ...                ...                ...                    ...
post(&forks[4])    post(&forks[0])    post(&forks[1])    post(&forks[2])        post(&forks[3])
post(&forks[0])    post(&forks[1])    post(&forks[2])    post(&forks[3])        post(&forks[4])
post(&eating)      post(&eating)      post(&eating)      post(&eating)          post(&eating)
```

The Dining Philosophers Problem
Solutions 2: Illustration

```
Philosopher 1       Philosopher 2       Philosopher 3       Philosopher 4       Philosopher 5

wait(&eating)       wait(&eating)       wait(&eating)       wait(&eating)       wait(&eating)-1=>-2
wait(&forks[4])     wait(&forks[0])     wait(&forks[1])     wait(&forks[2])     wait(&forks[3])
wait(&forks[0])     wait(&forks[1])     wait(&forks[2])     wait(&forks[3])     wait(&forks[4])
...                 ...                 ...                 ...                 ...
// eating           // eating           // eating           // eating           // eating
...                 ...                 ...                 ...                 ...
post(&forks[4])     post(&forks[0])     post(&forks[1])     post(&forks[2])     post(&forks[3])
post(&forks[0])     post(&forks[1])     post(&forks[2])     post(&forks[3])     post(&forks[4])
post(&eating)       post(&eating)       post(&eating)       post(&eating)       post(&eating)
```

The Dining Philosophers Problem

Solutions 2: Illustration



```
Philosopher 1         Philosopher 2        Philosopher 3        Philosopher 4        Philosopher 5

wait(&eating)-2=>-3   wait(&eating)        wait(&eating)        wait(&eating)        wait(&eating)
wait(&forks[4])       wait(&forks[0])      wait(&forks[1])      wait(&forks[2])      wait(&forks[3])
wait(&forks[0])       wait(&forks[1])      wait(&forks[2])      wait(&forks[3])      wait(&forks[4])
...                   ...                  ...                  ...                  ...
// eating             // eating            // eating            // eating            // eating
...                   ...                  ...                  ...                  ...
post(&forks[4])       post(&forks[0])      post(&forks[1])      post(&forks[2])      post(&forks[3])
post(&forks[0])       post(&forks[1])      post(&forks[2])      post(&forks[3])      post(&forks[4])
post(&eating)         post(&eating)        post(&eating)        post(&eating)        post(&eating)
```

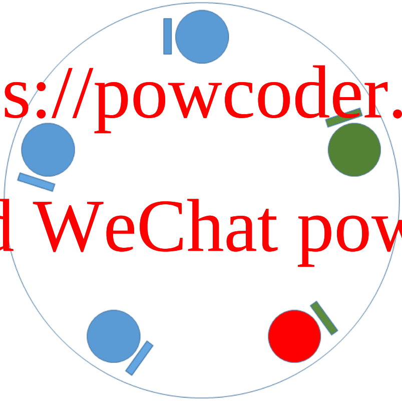
The Dining Philosophers Problem

Solution 3: Maximum Parallelism

- A **more sophisticated solution** is necessary to allow **maximum parallelism**
- The solution uses:
  - `state[N]`: one **state variable** for every philosopher (`THINKING`, `HUNGRY`, `EATING`)
  - `phil[N]`: one **semaphore per *philosopher*** (i.e., **not forks**, **initialised to 0**)
    - The philosopher **goes to sleep** if one of his/her neighbours are eating
    - The **neighbours wake up the philosopher** if they have finished eating
  - `sync`: one **semaphore/mutex** to enforce **mutual exclusion** of the critical section (while updating the **states**)

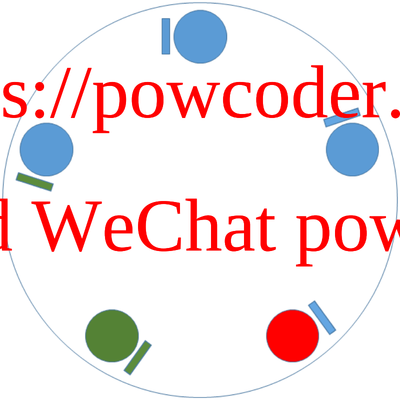
The Dining Philosophers Problem
Solution 3: Maximum Parallelism

- A philosopher can only **start eating** if his/her **neighbours are not eating**



The Dining Philosophers Problem
Solution 3: Maximum Parallelism

- A philosopher can only **start eating** if his/her **neighbours are not eating**



The Dining Philosophers Problem

Solution 3: Maximum Parallelism

```
#define N 5
#define THINKING 1
#define HUNGRY 2
#define EATING 3

int state[N] = {THINKING, THINKING, THINKING, THINKING, THINKING};
sem_t phil[N];  // sends philosopher to sleep
sem_t sync;


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

The Dining Philosophers Problem
Solution 3: Maximum Parallelism

```
void take_forks(int i)
{
  sem_wait(&sync);
  state[i] = HUNGRY;
  test(i);
  sem_post(&sync);
  sem_wait(&phil[i]);
}

void test(int i)
{
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  if(state[i] == HUNGRY
    && state[left] != EATING
    && state[right] != EATING) {
    state[i] = EATING;
    sem_post(&phil[i]);
  }
}
```

The Dining Philosophers Problem
Solution 3: Maximum Parallelism

```
void put_forks(int i)
{
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  sem_wait(&sync);
  state[i] = THINKING;
  test(left);
  test(right);
  sem_post(&sync);
}

void test(int i)
{
  int left = (i + N - 1) % N;
  int right = (i + 1) % N;
  if(state[i] == HUNGRY
    && state[left] != EATING
    && state[right] != EATING) {
    state[i] = EATING;
    sem_post(&phil[i]);
  }
}
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

```
Philosopher 2
(left = 1, right = 3)

wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT){
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT){
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

```
Philosopher 3
(left = 2, right = 4)

wait(&sync) // 1 => 0
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
wait(&phil[3])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT){
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT){
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

```
Philosopher 4
(left = 3, right = 5)

wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT){
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT){
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem
Solution 3: Maximum Parallelism

```
Philosopher 2
(left = 1, right = 3)

wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
   && state[1]!=EAT
   && state[3]!=EAT){
   state[2]=EAT
   post(&phil[2])
}
post(&sync)
wait(&phil[2])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
   && state[5]!=EAT
   && state[2]!=EAT){
   state[1]=EAT
   post(&phil[1])
}
if(state[3]==HUNGRY
   && state[2]!=EAT
   && state[4]!=EAT){
   state[3]=EAT
   post(&phil[3])
}
post(&sync)
```

```
Philosopher 3
(left = 2, right = 4)

wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
   && state[2]!=EAT
   && state[4]!=EAT){
   state[3]=EAT
   post(&phil[3])
}
post(&sync)
wait(&phil[3])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
   && state[1]!=EAT
   && state[3]!=EAT){
   state[2]=EAT
   post(&phil[2])
}
if(state[4]==HUNGRY
   && state[3]!=EAT
   && state[5]!=EAT){
   state[4]=EAT
   post(&phil[4])
}
post(&sync)
```

```
Philosopher 4
(left = 3, right = 5)

wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
   && state[3]!=EAT
   && state[5]!=EAT){
   state[4]=EAT
   post(&phil[4])
}
post(&sync)
wait(&phil[4])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
   && state[2]!=EAT
   && state[4]!=EAT){
   state[3]=EAT
   post(&phil[3])
}
if(state[5]==HUNGRY
   && state[4]!=EAT
   && state[1]!=EAT){
   state[5]=EAT
   post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

```
Philosopher 2
(left = 1, right = 3)

wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT){
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT){
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

```
Philosopher 3
(left = 2, right = 4)

wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])  // 0 => 1
}
post(&sync)
wait(&phil[3])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT){
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT){
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

```
Philosopher 4
(left = 3, right = 5)

wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT){
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT){
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem
Solution 3: Maximum Parallelism

```
Philosopher 2
(left = 1, right = 3)

wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT){
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT){
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

```
Philosopher 3
(left = 2, right = 4)

wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
post(&sync)  // 0 -> 1
wait(&phil[3])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT){
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT){
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

```
Philosopher 4
(left = 3, right = 5)

wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT){
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT){
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

```
Philosopher 2
(left = 1, right = 3)

wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT){
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT){
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

```
Philosopher 3
(left = 2, right = 4)

wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])   // 0 => 1
}
post(&sync)
wait(&phil[3])   // 1 => 0

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT){
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT){
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

```
Philosopher 4
(left = 3, right = 5)

wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT){
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT){
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

```
Philosopher 2
(left = 1, right = 3)

wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT){
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT){
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

```
Philosopher 3
(left = 2, right = 4)

wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
wait(&phil[3])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT){
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT){
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

```
Philosopher 4
(left = 3, right = 5)

wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT){
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT){
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

```
Philosopher 2
(left = 1, right = 3)

wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT){
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT){
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

```
Philosopher 3
(left = 2, right = 4)

wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
wait(&phil[3])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync) // 1 => 0
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT){
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT){
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

```
Philosopher 4
(left = 3, right = 5)

wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT){
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT){
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

```
Philosopher 2
(left = 1, right = 3)

wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT){
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT){
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

```
Philosopher 3
(left = 2, right = 4)

wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
wait(&phil[3])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT){
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT){
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

```
Philosopher 4
(left = 3, right = 5)

wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT){
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT){
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem
Solution 3: Maximum Parallelism

```
Philosopher 2
(left = 1, right = 3)

wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
   && state[1]!=EAT
   && state[3]!=EAT){
   state[2]=EAT
   post(&phil[2])
}
post(&sync)
wait(&phil[2])   // assume == -1 (wakeup)

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
   && state[5]!=EAT
   && state[2]!=EAT){
   state[1]=EAT
   post(&phil[1])
}
if(state[3]==HUNGRY
   && state[2]!=EAT
   && state[4]!=EAT){
   state[3]=EAT
   post(&phil[3])
}
post(&sync)
```

```
Philosopher 3
(left = 2, right = 4)

wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
   && state[2]!=EAT
   && state[4]!=EAT){
   state[3]=EAT
   post(&phil[3])
}
post(&sync)
wait(&phil[3])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
   && state[1]!=EAT
   && state[3]!=EAT){
   state[2]=EAT
   post(&phil[2])   // -1 => 0
}
if(state[4]==HUNGRY
   && state[3]!=EAT
   && state[5]!=EAT){
   state[4]=EAT
   post(&phil[4])
}
post(&sync)
```

```
Philosopher 4
(left = 3, right = 5)

wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
   && state[3]!=EAT
   && state[5]!=EAT){
   state[4]=EAT
   post(&phil[4])
}
post(&sync)
wait(&phil[4])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
   && state[2]!=EAT
   && state[4]!=EAT){
   state[3]=EAT
   post(&phil[3])
}
if(state[5]==HUNGRY
   && state[4]!=EAT
   && state[1]!=EAT){
   state[5]=EAT
   post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

```
Philosopher 2
(left = 1, right = 3)

wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT){
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT){
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

```
Philosopher 3
(left = 2, right = 4)

wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
wait(&phil[3])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT){
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT){
  state[4]=EAT
  post(&phil[4])   // -1 => 0
}
post(&sync)
```

```
Philosopher 4
(left = 3, right = 5

wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT){
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4]) // assume -1 (

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT){
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

```
Philosopher 2
(left = 1, right = 3)

wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT){
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT){
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

```
Philosopher 3
(left = 2, right = 4)

wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
wait(&phil[3])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT){
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT){
  state[4]=EAT
  post(&phil[4])
}
post(&sync)   // 0 => 1
```

```
Philosopher 4
(left = 3, right = 5)

wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT){
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT){
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

```
Philosopher 2
(left = 1, right = 3)

wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT){
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT){
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

```
Philosopher 3
(left = 2, right = 4)

wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
wait(&phil[3])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT){
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT){
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

```
Philosopher 4
(left = 3, right = 5)

wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT){
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT){
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

```
Philosopher 2
(left = 1, right = 3)

wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT){
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT){
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

```
Philosopher 3
(left = 2, right = 4)

wait(&sync) // 1 => 0
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
wait(&phil[3])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT){
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT){
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

```
Philosopher 4
(left = 3, right = 5)

wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT){
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT){
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem
Solution 3: Maximum Parallelism

```
Philosopher 2
(left = 1, right = 3)

wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
   && state[1]!=EAT
   && state[3]!=EAT){
   state[2]=EAT
   post(&phil[2])
}
post(&sync)
wait(&phil[2])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
   && state[5]!=EAT
   && state[2]!=EAT){
   state[1]=EAT
   post(&phil[1])
}
if(state[3]==HUNGRY
   && state[2]!=EAT
   && state[4]!=EAT){
   state[3]=EAT
   post(&phil[3])
}
post(&sync)
```

```
Philosopher 3
(left = 2, right = 4)

wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
   && state[2]!=EAT
   && state[4]!=EAT){
   state[3]=EAT
   post(&phil[3])
}
post(&sync)
wait(&phil[3])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
   && state[1]!=EAT
   && state[3]!=EAT){
   state[2]=EAT
   post(&phil[2])
}
if(state[4]==HUNGRY
   && state[3]!=EAT
   && state[5]!=EAT){
   state[4]=EAT
   post(&phil[4])
}
post(&sync)
```

```
Philosopher 4
(left = 3, right = 5)

wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
   && state[3]!=EAT
   && state[5]!=EAT){
   state[4]=EAT
   post(&phil[4])
}
post(&sync)
wait(&phil[4])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
   && state[2]!=EAT
   && state[4]!=EAT){
   state[3]=EAT
   post(&phil[3])
}
if(state[5]==HUNGRY
   && state[4]!=EAT
   && state[1]!=EAT){
   state[5]=EAT
   post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

```
Philosopher 2
(left = 1, right = 3)

wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
   && state[1]!=EAT
   && state[3]!=EAT){
   state[2]=EAT
   post(&phil[2])
}
post(&sync)
wait(&phil[2])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
   && state[5]!=EAT
   && state[2]!=EAT){
   state[1]=EAT
   post(&phil[1])
}
if(state[3]==HUNGRY
   && state[2]!=EAT
   && state[4]!=EAT){
   state[3]=EAT
   post(&phil[3])
}
post(&sync)
```

```
Philosopher 3
(left = 2, right = 4)

wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
   && state[2]!=EAT
   && state[4]!=EAT){
   state[3]=EAT
   post(&phil[3])
}
post(&sync)
wait(&phil[3])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
   && state[1]!=EAT
   && state[3]!=EAT){
   state[2]=EAT
   post(&phil[2])
}
if(state[4]==HUNGRY
   && state[3]!=EAT
   && state[5]!=EAT){
   state[4]=EAT
   post(&phil[4])
}
post(&sync)
```

```
Philosopher 4
(left = 3, right = 5)

wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
   && state[3]!=EAT
   && state[5]!=EAT){
   state[4]=EAT
   post(&phil[4])
}
post(&sync)
wait(&phil[4])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
   && state[2]!=EAT
   && state[4]!=EAT){
   state[3]=EAT
   post(&phil[3])
}
if(state[5]==HUNGRY
   && state[4]!=EAT
   && state[1]!=EAT){
   state[5]=EAT
   post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

```
Philosopher 2
(left = 1, right = 3)

wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
   && state[1]!=EAT
   && state[3]!=EAT){
   state[2]=EAT
   post(&phil[2])
}
post(&sync)
wait(&phil[2])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
   && state[5]!=EAT
   && state[2]!=EAT){
   state[1]=EAT
   post(&phil[1])
}
if(state[3]==HUNGRY
   && state[2]!=EAT
   && state[4]!=EAT){
   state[3]=EAT
   post(&phil[3])
}
post(&sync)
```

```
Philosopher 3
(left = 2, right = 4)

wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
   && state[2]!=EAT
   && state[4]!=EAT){
   state[3]=EAT
   post(&phil[3])
}
post(&sync)  // 0 -> 1
wait(&phil[3])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
   && state[1]!=EAT
   && state[3]!=EAT){
   state[2]=EAT
   post(&phil[2])
}
if(state[4]==HUNGRY
   && state[3]!=EAT
   && state[5]!=EAT){
   state[4]=EAT
   post(&phil[4])
}
post(&sync)
```

```
Philosopher 4
(left = 3, right = 5)

wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
   && state[3]!=EAT
   && state[5]!=EAT){
   state[4]=EAT
   post(&phil[4])
}
post(&sync)
wait(&phil[4])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
   && state[2]!=EAT
   && state[4]!=EAT){
   state[3]=EAT
   post(&phil[3])
}
if(state[5]==HUNGRY
   && state[4]!=EAT
   && state[1]!=EAT){
   state[5]=EAT
   post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

```
Philosopher 2
(left = 1, right = 3)

wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT){
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT){
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

```
Philosopher 3
(left = 2, right = 4)

wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
wait(&phil[3])  // 0 => -1 (sleeping)

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT){
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT){
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

```
Philosopher 4
(left = 3, right = 5)

wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT){
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT){
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

```
Philosopher 2
(left = 1, right = 3)

wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT){
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)  // 1 => 0
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT){
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

```
Philosopher 3
(left = 2, right = 4)

wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
wait(&phil[3])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT){
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT){
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

```
Philosopher 4
(left = 3, right = 5)

wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT){
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT){
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

```
Philosopher 2
(left = 1, right = 3)

wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT){
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT){
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

```
Philosopher 3
(left = 2, right = 4)

wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
wait(&phil[3])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT){
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT){
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

```
Philosopher 4
(left = 3, right = 5)

wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT){
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT){
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

```
Philosopher 2
(left = 1, right = 3)

wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT){
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT){
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
```

```
Philosopher 3
(left = 2, right = 4)

wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
wait(&phil[3])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT){
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT){
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

```
Philosopher 4
(left = 3, right = 5)

wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT){
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT){
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Dining Philosophers Problem

Solution 3: Maximum Parallelism

```
Philosopher 2
(left = 1, right = 3)

wait(&sync)
state[2]=HUNGRY
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT){
  state[2]=EAT
  post(&phil[2])
}
post(&sync)
wait(&phil[2])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[2] = THINK
// test neighbours
if(state[1]==HUNGRY
  && state[5]!=EAT
  && state[2]!=EAT){
  state[1]=EAT
  post(&phil[1])
}
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])  // -1 => 0
}
post(&sync)
```

```
Philosopher 3
(left = 2, right = 4)

wait(&sync)
state[3]=HUNGRY
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
post(&sync)
wait(&phil[3])  // wakeup

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[3] = THINK
// test neighbours
if(state[2]==HUNGRY
  && state[1]!=EAT
  && state[3]!=EAT){
  state[2]=EAT
  post(&phil[2])
}
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT){
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
```

```
Philosopher 4
(left = 3, right = 5)

wait(&sync)
state[4]=HUNGRY
if(state[4]==HUNGRY
  && state[3]!=EAT
  && state[5]!=EAT){
  state[4]=EAT
  post(&phil[4])
}
post(&sync)
wait(&phil[4])

// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT
// EAT EAT EAT EAT EAT

wait(&sync)
state[4] = THINK
// test neighbours
if(state[3]==HUNGRY
  && state[2]!=EAT
  && state[4]!=EAT){
  state[3]=EAT
  post(&phil[3])
}
if(state[5]==HUNGRY
  && state[4]!=EAT
  && state[1]!=EAT){
  state[5]=EAT
  post(&phil[5])
}
post(&sync)
```

The Readers – Writers Problem
Description

- Concurrent database processes are readers and/or writers, files, I/O devices, etc.
- **Reading** a record (variable) can happen **in parallel** without problems, **writing needs synchronisation** (i.e. exclusive access)

- **Different solutions** exist to the readers/writers problem
  - Solution 1: naive implementation with **limited parallelism**
  - Solution 2: **readers** receive **priority**: **no reader is kept waiting** (unless a writer already has access, **writers may starve**)
  - Solution 3: **writing** is performed **as soon as possibly** (**readers may starve**)

The Readers – Writers Problem

Solution 1: No Parallelism

```
void * reader(void * arg)
{
  while(1)
  {
    pthread_mutex_lock(&sync);
    printf("reading record\n");
    pthread_mutex_unlock(&sync);
  }
}

void * writer(void * writer)
{
  while(1)
  {
    pthread_mutex_lock(&sync);
    printf("writing\n");
    pthread_mutex_unlock(&sync);
  }
}
```

The Readers – Writers Problem
Solution 2: Readers First

- Solution 1: prevents **parallel reading**
- Solution 2: **allows parallel reading**
- A correct implementation of solution 2 requires:

  - `iReadCount`: an integer tracking the number of readers
    - If `iReadCount > 0`: writers are blocked (`sem_wait(rwSync)`)
    - If `iReadCount == 0`: writers are released (`sem_post(rwSync)`)
    - If already writing, readers must wait

  - `sync`: a mutex for mutual exclusion of `iReadCount`
  - `rwSync`: a semaphore that **synchronises the readers and writers**, set by the **first/last reader**

The Readers – Writers Problem

Solution 2: Readers First

```
void * reader(void * arg)
{
  while(1)
  {
    pthread_mutex_lock(&sync);
    iReadCount++;
    if(iReadCount == 1)
      sem_wait(&rwSync);
    pthread_mutex_unlock(&sync);

    printf("reading record\n");

    pthread_mutex_lock(&sync);
    iReadCount--;
    if(iReadCount == 0)
      sem_post(&rwSync);
    pthread_mutex_unlock(&sync);
  }
}
```

```
void * writer(void * writer)
{
  while(1)
  {
    sem_wait(&rwSync);
    printf("writing\n");
    sem_post(&rwSync);
  }
}
```

The Readers – Writers Problem

Solution 2: Readers First

```
void * reader(void * arg)
{
  while(1)
  {
    pthread_mutex_lock(&sync);
    iReadCount++;
    if(iReadCount == 1)
      sem_wait(&rwSync);
    pthread_mutex_unlock(&sync);

    printf("reading record\n");

    pthread_mutex_lock(&sync);
    iReadCount--;
    if(iReadCount == 0)
      sem_post(&rwSync);
    pthread_mutex_unlock(&sync);
  }
}
```

```
void * writer(void * writer)
{
  while(1)
  {
    sem_wait(&rwSync);
    printf("writing\n");
    sem_post(&rwSync);
  }
}
```

The Readers – Writers Problem

Solution 2: Readers First

```
void * reader(void * arg)
{
  while(1)
  {
    pthread_mutex_lock(&sync); // 1=>0
    iReadCount++;
    if(iReadCount == 1)
      sem_wait(&rwSync);
    pthread_mutex_unlock(&sync);

    printf("reading record\n");

    pthread_mutex_lock(&sync);
    iReadCount--;
    if(iReadCount == 0)
      sem_post(&rwSync);
    pthread_mutex_unlock(&sync);
  }
}
```

```
void * writer(void * writer)
{
  while(1)
  {
    sem_wait(&rwSync);
    printf("writing\n");
    sem_post(&rwSync);
  }
}
```

The Readers – Writers Problem
Solution 2: Readers First

```
void * reader(void * arg)
{
  while(1)
  {
    pthread_mutex_lock(&sync);
    iReadCount++;  // 0=>1
    if(iReadCount == 1)
      sem_wait(&rwSync);
    pthread_mutex_unlock(&sync);

    printf("reading record\n");

    pthread_mutex_lock(&sync);
    iReadCount--;
    if(iReadCount == 0)
      sem_post(&rwSync);
    pthread_mutex_unlock(&sync);
  }
}
```

```
void * writer(void * writer)
{
  while(1)
  {
    sem_wait(&rwSync);
    printf("writing\n");
    sem_post(&rwSync);
  }
}
```

The Readers – Writers Problem

Solution 2: Readers First

```
void * reader(void * arg)
{
  while(1)
  {
    pthread_mutex_lock(&sync);
    iReadCount++;
    if(iReadCount == 1)
      sem_wait(&rwSync);   // 1=>0
    pthread_mutex_unlock(&sync);

    printf("reading record\n");

    pthread_mutex_lock(&sync);
    iReadCount--;
    if(iReadCount == 0)
      sem_post(&rwSync);
    pthread_mutex_unlock(&sync);
  }
}
```

```
void * writer(void * writer)
{
  while(1)
  {
    sem_wait(&rwSync);
    printf("writing\n");
    sem_post(&rwSync);
  }
}
```

The Readers – Writers Problem

Solution 2: Readers First

```
void * reader(void * arg)
{
  while(1)
  {
    pthread_mutex_lock(&sync);
    iReadCount++;
    if(iReadCount == 1)
      sem_wait(&rwSync);
    pthread_mutex_unlock(&sync); // 0=>1

    printf("reading record\n");

    pthread_mutex_lock(&sync);
    iReadCount--;
    if(iReadCount == 0)
      sem_post(&rwSync);
    pthread_mutex_unlock(&sync);
  }
}
```

```
void * writer(void * writer)
{
  while(1)
  {
    sem_wait(&rwSync);
    printf("writing\n");
    sem_post(&rwSync);
  }
}
```

The Readers – Writers Problem
Solution 2: Readers First

```
void * reader(void * arg)
{
  while(1)
  {
    pthread_mutex_lock(&sync);
    iReadCount++;
    if(iReadCount == 1)
      sem_wait(&rwSync);
    pthread_mutex_unlock(&sync);

    printf("reading record\n");

    pthread_mutex_lock(&sync);
    iReadCount--;
    if(iReadCount == 0)
      sem_post(&rwSync);
    pthread_mutex_unlock(&sync);
  }
}
```

```
void * writer(void * writer)
{
  while(1)
  {
    sem_wait(&rwSync);   // 0=>-1
    printf("writing\n");
    sem_post(&rwSync);
  }
}
```

The Readers – Writers Problem
Solution 2: Readers First

```
void * reader(void * arg)
{
  while(1)
  {
    pthread_mutex_lock(&sync);
    iReadCount++;
    if(iReadCount == 1)
      sem_wait(&rwSync);
    pthread_mutex_unlock(&sync);

    printf("reading record\n");

    pthread_mutex_lock(&sync);
    iReadCount--;
    if(iReadCount == 0)
      sem_post(&rwSync);
    pthread_mutex_unlock(&sync);
  }
}
```

```
void * writer(void * writer)
{
  while(1)
  {
    sem_wait(&rwSync);
    printf("writing\n");
    sem_post(&rwSync);
  }
}
```

The Readers – Writers Problem
Solution 2: Readers First

```
void * reader(void * arg)
{
  while(1)
  {
    pthread_mutex_lock(&sync);
    iReadCount++;
    if(iReadCount == 1)
      sem_wait(&rwSync);
    pthread_mutex_unlock(&sync);

    printf("reading record\n");

    pthread_mutex_lock(&sync); // 1=>0
    iReadCount--;
    if(iReadCount == 0)
      sem_post(&rwSync);
    pthread_mutex_unlock(&sync);
  }
}
```

```
void * writer(void * writer)
{
  while(1)
  {
    sem_wait(&rwSync);
    printf("writing\n");
    sem_post(&rwSync);
  }
}
```

The Readers – Writers Problem

Solution 2: Readers First

```c
void * reader(void * arg)
{
  while(1)
  {
    pthread_mutex_lock(&sync);
    iReadCount++;
    if(iReadCount == 1)
      sem_wait(&rwSync);
    pthread_mutex_unlock(&sync);

    printf("reading record\n");

    pthread_mutex_lock(&sync);
    iReadCount--;  // 1=>0
    if(iReadCount == 0)
      sem_post(&rwSync);
    pthread_mutex_unlock(&sync);
  }
}
```

```c
void * writer(void * writer)
{
  while(1)
  {
    sem_wait(&rwSync);
    printf("writing\n");
    sem_post(&rwSync);
  }
}
```

The Readers – Writers Problem
Solution 2: Readers First

```
void * reader(void * arg)
{
  while(1)
  {
    pthread_mutex_lock(&sync);
    iReadCount++;
    if(iReadCount == 1)
      sem_wait(&rwSync);
    pthread_mutex_unlock(&sync);

    printf("reading record\n");

    pthread_mutex_lock(&sync);
    iReadCount--;
    if(iReadCount == 0)
      sem_post(&rwSync);  // -1=>0
    pthread_mutex_unlock(&sync);
  }
}
```

```
void * writer(void * writer)
{
  while(1)
  {
    sem_wait(&rwSync);  // wakeup
    printf("writing\n");
    sem_post(&rwSync);
  }
}
```

The Readers – Writers Problem

Solution 2: Readers First

```
void * reader(void * arg)
{
  while(1)
  {
    pthread_mutex_lock(&sync);
    iReadCount++;
    if(iReadCount == 1)
      sem_wait(&rwSync);
    pthread_mutex_unlock(&sync);

    printf("reading record\n");

    pthread_mutex_lock(&sync);
    iReadCount--;
    if(iReadCount == 0)
      sem_post(&rwSync);
    pthread_mutex_unlock(&sync);   // 0=>1
  }
}
```

```
void * writer(void * writer)
{
  while(1)
  {
    sem_wait(&rwSync);
    printf("writing\n");
    sem_post(&rwSync);
  }
}
```

The Readers – Writers Problem

Solution 2: Readers First

```
void * reader(void * arg)
{
  while(1)
  {
    pthread_mutex_lock(&sync);
    iReadCount++;
    if(iReadCount == 1)
      sem_wait(&rwSync);
    pthread_mutex_unlock(&sync);

    printf("reading record\n");

    pthread_mutex_lock(&sync);
    iReadCount--;
    if(iReadCount == 0)
      sem_post(&rwSync);
    pthread_mutex_unlock(&sync);
  }
}
```

```
void * writer(void * writer)
{
  while(1)
  {
    sem_wait(&rwSync);
    printf("writing\n");
    sem_post(&rwSync);
  }
}
```

The Readers – Writers Problem
Solution 2: Readers First

```
void * reader(void * arg)
{
  while(1)
  {
    pthread_mutex_lock(&sync);
    iReadCount++;
    if(iReadCount == 1)
      sem_wait(&rwSync);
    pthread_mutex_unlock(&sync);

    printf("reading record\n");

    pthread_mutex_lock(&sync);
    iReadCount--;
    if(iReadCount == 0)
      sem_post(&rwSync);
    pthread_mutex_unlock(&sync);
  }
}
```

```
void * writer(void * writer)
{
  while(1)
  {
    sem_wait(&rwSync);
    printf("writing\n");
    sem_post(&rwSync);   // 0=>1
  }
}
```

The Readers – Writers Problem

Solution 2: Readers First

```
void * reader(void * arg)
{
  while(1)
  {
    pthread_mutex_lock(&sync);
    iReadCount++;
    if(iReadCount == 1)
      sem_wait(&rwSync);
    pthread_mutex_unlock(&sync);

    printf("reading record\n");

    pthread_mutex_lock(&sync);
    iReadCount--;
    if(iReadCount == 0)
      sem_post(&rwSync);
    pthread_mutex_unlock(&sync);
  }
}
```

```
void * writer(void * writer)
{
  while(1)
  {
    sem_wait(&rwSync); // 1=>0
    printf("writing\n");
    sem_post(&rwSync);
  }
}
```

The Readers – Writers Problem

Solution 2: Readers First

```
void * reader(void * arg)
{
  while(1)
  {
    pthread_mutex_lock(&sync); // 1=>0
    iReadCount++;
    if(iReadCount == 1)
      sem_wait(&rwSync);
    pthread_mutex_unlock(&sync);

    printf("reading record\n");

    pthread_mutex_lock(&sync);
    iReadCount--;
    if(iReadCount == 0)
      sem_post(&rwSync);
    pthread_mutex_unlock(&sync);
  }
}
```

```
void * writer(void * writer)
{
  while(1)
  {
    sem_wait(&rwSync);
    printf("writing\n");
    sem_post(&rwSync);
  }
}
```

The Readers – Writers Problem
Solution 2: Readers First

```
void * reader(void * arg)
{
  while(1)
  {
    pthread_mutex_lock(&sync);
    iReadCount++; // 0=>1
    if(iReadCount == 1)
      sem_wait(&rwSync);
    pthread_mutex_unlock(&sync);

    printf("reading record\n");

    pthread_mutex_lock(&sync);
    iReadCount--;
    if(iReadCount == 0)
      sem_post(&rwSync);
    pthread_mutex_unlock(&sync);
  }
}
```

```
void * writer(void * writer)
{
  while(1)
  {
    sem_wait(&rwSync);
    printf("writing\n");
    sem_post(&rwSync);
  }
}
```

The Readers – Writers Problem

Solution 2: Readers First

```
void * reader(void * arg)
{
  while(1)
  {
    pthread_mutex_lock(&sync);
    iReadCount++;
    if(iReadCount == 1)
      sem_wait(&rwSync); // 0=>-1 (sleep)
    pthread_mutex_unlock(&sync);

    printf("reading record\n");

    pthread_mutex_lock(&sync);
    iReadCount--;
    if(iReadCount == 0)
      sem_post(&rwSync);
    pthread_mutex_unlock(&sync);
  }
}
```

```
void * writer(void * writer)
{
  while(1)
  {
    sem_wait(&rwSync);
    printf("writing\n");
    sem_post(&rwSync);
  }
}
```

The Readers – Writers Problem

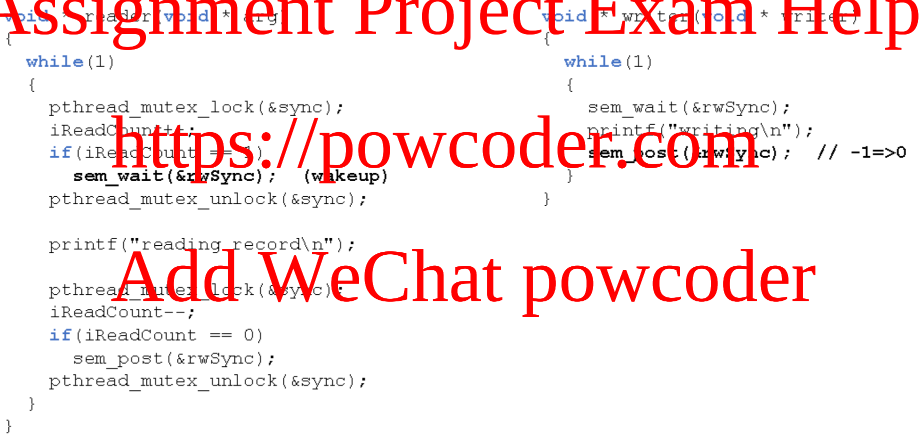
Solution 2: Readers First

```c
void * reader(void * arg)
{
  while(1)
  {
    pthread_mutex_lock(&sync);
    iReadCount++;
    if(iReadCount == 1)
      sem_wait(&rwSync);
    pthread_mutex_unlock(&sync);

    printf("reading record\n");

    pthread_mutex_lock(&sync);
    iReadCount--;
    if(iReadCount == 0)
      sem_post(&rwSync);
    pthread_mutex_unlock(&sync);
  }
}
```

```c
void * writer(void * writer)
{
  while(1)
  {
    sem_wait(&rwSync);
    printf("writing\n");
    sem_post(&rwSync);
  }
}
```

The Readers – Writers Problem

Solution 2: Readers First



```
void * reader(void * arg)
{
  while(1)
  {
    pthread_mutex_lock(&sync);
    iReadCount++;
    if(iReadCount == 1)
      sem_wait(&rwSync);   (wakeup)
    pthread_mutex_unlock(&sync);

    printf("reading record\n");

    pthread_mutex_lock(&sync);
    iReadCount--;
    if(iReadCount == 0)
      sem_post(&rwSync);
    pthread_mutex_unlock(&sync);
  }
}
```

```
void * writer(void * writer)
{
  while(1)
  {
    sem_wait(&rwSync);
    printf("writing\n");
    sem_post(&rwSync);   // -1=>0
  }
}
```

The Readers – Writers Problem

Solution 2: Readers First

```
void * reader(void * arg)
{
  while(1)
  {
    pthread_mutex_lock(&sync);
    iReadCount++;
    if(iReadCount == 1)
      sem_wait(&rwSync);
    pthread_mutex_unlock(&sync);

    printf("reading record\n");

    pthread_mutex_lock(&sync);
    iReadCount--;
    if(iReadCount == 0)
      sem_post(&rwSync);
    pthread_mutex_unlock(&sync);
  }
}
```

```
void * writer(void * writer)
{
  while(1)
  {
    sem_wait(&rwSync);
    printf("writing\n");
    sem_post(&rwSync);
  }
}
```

- **Dining philosophers** with **improved parallelism** and **maximum parallelism**
- Readers/writers problem
  - Solution with **limited/no parallelism**
  - Solution with **priority for the readers**