Classical Problems of Synchronization

A Quick recap on Semaphore

- Definition of the **`wait()` operation**

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

- Definition of the **`signal()` operation**

```
signal(S) {
    S++;
}
```

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- The producer and consumer process share the following data structures:

  int n   [n buffers, each can hold one item]

  semaphore mutex =1

  semaphore full = 0

  semaphore empty = n

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {
     ...
      /* produce an item in next_produced */
     ...
   wait(empty);
    wait(mutex);
      ...
      /* add next_produced to the buffer */
      ...
   signal(mutex);
    signal(full);
} while (true);
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
do {
     wait(full);
     wait(mutex);
        . . .
    /* remove an item from buffer to next_consumed */
        . . .
    signal(mutex);
     signal(empty);
        . . .
    /* consume the item in next_consumed */
        . . .
  } while (true);
```

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do  {
        //   produce an item
         wait (empty);
         wait (mutex);
           //  add the item to the buffer
        signal (mutex);
         signal (full);
   } while (TRUE);
```

- The structure of the consumer process

```
do {
     wait (full);
     wait (mutex);
   //remove an item from buffer
      signal (mutex);
      signal (empty);
    //  consume the item
 } while (TRUE);
```

Readers-Writers Problem

- A database is shared among a number of concurrent processes
  - Readers – only read the data set; they do **not** perform any updates
  - Writers – can both read and write
- Problem –
  - allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time

Readers-Writers Problem Variations

- ***First*** variation – no reader kept waiting unless writer has already obtained permission to use shared object
- ***Second*** variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

- Shared Data

  Database

  semaphore **rw_mutex** = 1

  semaphore **mutex** = 1

  int **read_count** = 0

Readers-Writers Problem (Cont.)

- The structure of a writer process

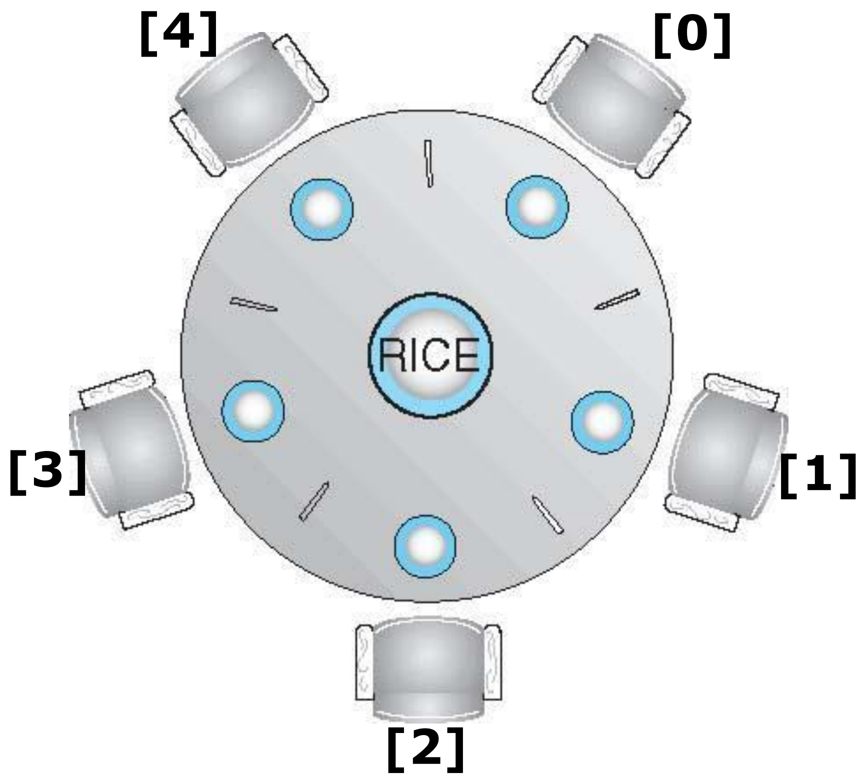
```
do {
        wait(rw_mutex);
            ...
        /* writing is performed */
            ...
    signal(rw_mutex);
} while (true);
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
        wait(mutex);
        read_count++;
        if (read_count == 1)
           wait(rw_mutex);
        signal(mutex);
       ...
        /* reading is performed */
       ...
     wait(mutex);
        read count--;
        if (read_count == 0)
           signal(rw_mutex);
        signal(mutex);
   } while (true);
```

Dining-Philosophers Problem



- Philosophers spend their lives thinking and eating

- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i* :

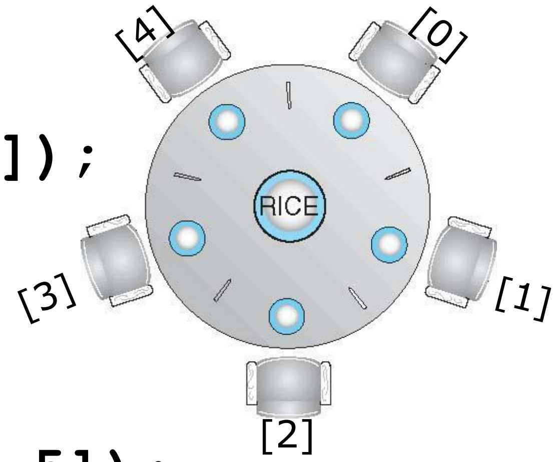
```
do {
    wait (chopstick[i]);
    wait (chopStick[(i + 1) % 5]);

             // eat

    signal (chopstick[i]);
    signal (chopstick[(i + 1) % 5]);

             // think

} while (TRUE);
```



- What is the problem with this algorithm?

Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section).
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

Problems with Semaphores

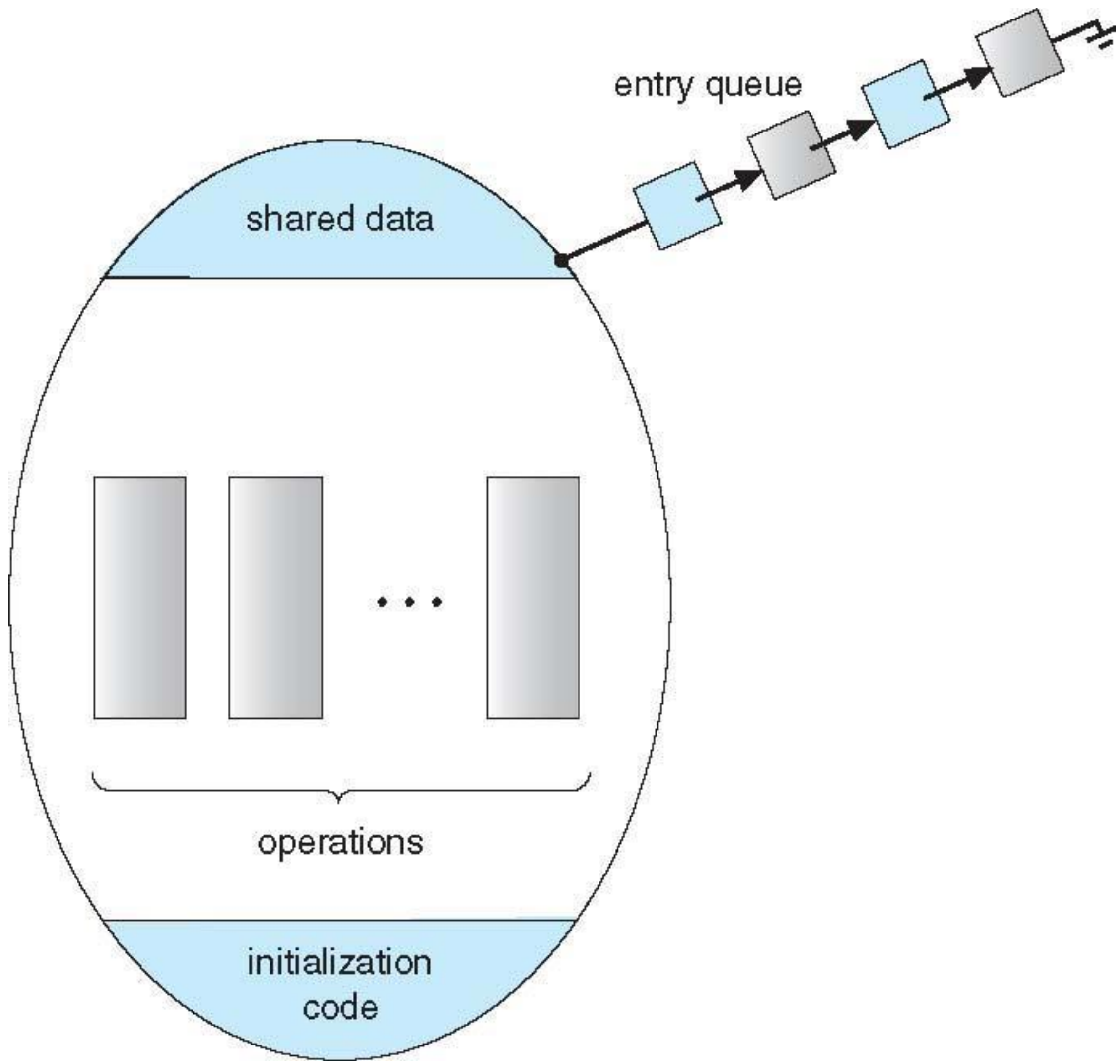
- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation are possible

Monitors

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- ***Abstract data type***, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

Schematic view of a Monitor



Syntax of a Monitor

```
monitor monitor-name
{
  // shared variable declarations

  procedure P1 (…) { …. }

  procedure Pn (…) {……}

     initialization code (…) { … }
  }
}
```

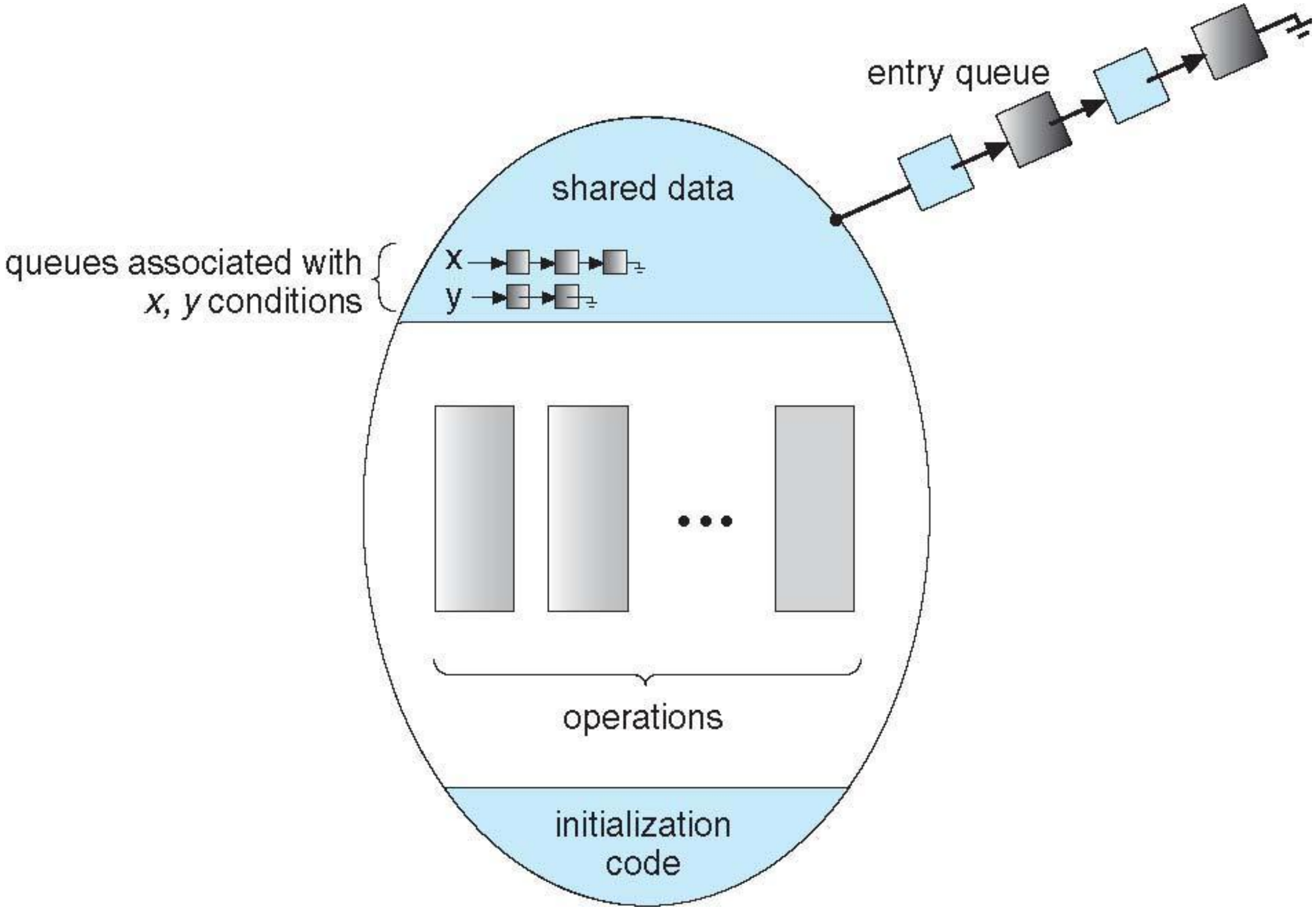
Condition Variables

- Additional synchronization mechanism: **condition** construct

```
condition x, y;
```

- Two operations on a condition variable:
  - `x.wait()` – a process that invokes the operation is suspended until `x.signal()`
  - `x.signal()` – resumes one of processes (if any) that invoked `x.wait()`
    - If no process is suspended, then signal () has no effect on the variable

Monitor with Condition Variables



Condition Variables Choices

- If process P invokes `x.signal()`, with Q in `x.wait()` state, what should happen next?
  - If Q is resumed, then P must wait

- Options include
  - **Signal and wait** – P waits until Q leaves monitor or waits for another condition
  - **Signal and continue** – Q waits until P leaves the monitor or waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed

Monitor Solution to Dining Philosophers

- Following data structures are used

```
enum {THINKING, HUNGRY, EATING} state [5];

condition self[5];
```

Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
  enum {THINKING, HUNGRY, EATING} state[5] ;
  condition self[5];

  void pickup (int i) {
         state[i] = HUNGRY;
         test(i);
         if (state[i] != EATING)
     self[i].wait;
   }

  void putdown (int i) {
        state[i] = THINKING;
        test((i + 4) % 5);  // test left and
        test((i + 1) % 5);  // right neighbors
   }
```

Solution to Dining Philosophers (Cont.)

```
void test (int i) {
        if ((state[(i + 4) % 5] != EATING) &&
          (state[i] == HUNGRY) &&
          (state[(i + 1) % 5] != EATING) ) {
               state[i] = EATING ;
          self[i].signal () ;
             }
 }

initialization_code() {
     for (int i = 0; i < 5; i++)
          state[i] = THINKING;
     }
}
```

Solution to Dining Philosophers (Cont.)

- Each philosopher *i* invokes the operations **`pickup()`** and **`putdown()`** in the following sequence:

```
        DiningPhilosophers.pickup(i);
......
             eat
......
        DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible