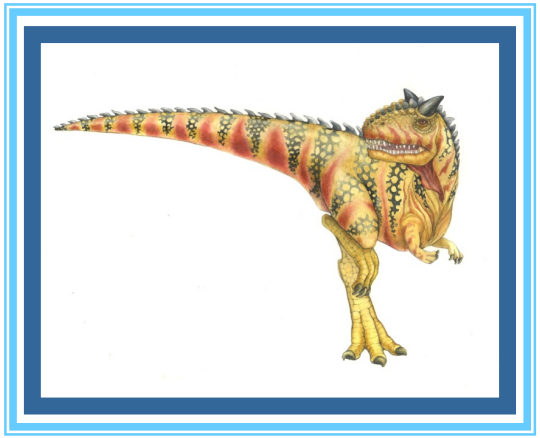
Chapter 7: Synchronization Examples



Classical Problems of Synchronization

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

Recall: Semaphore

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

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

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

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

Readers-Writers Problem

- A data set 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
- If two readers access the shared data simultaneously, no adverse affects
- If writer and some other thread (reader or writer) access data simultaneously, problem arises
- Ensure writer has exclusive access to data

Readers-Writers Problem

- Shared Data
  - Data set
  - Integer **`read_count`** initialized to 0
    - Keep count of how many processes are currently reading the data
  - Semaphore **`rw_mutex`** initialized to 1
    - Semaphore common to both reader and writer
  - Semaphore **`mutex`** initialized to 1
    - Semaphore to ensure mutual exclusion when **`read_count`** is updated

Readers-Writers Problem (Cont.)

- The structure of a writer process

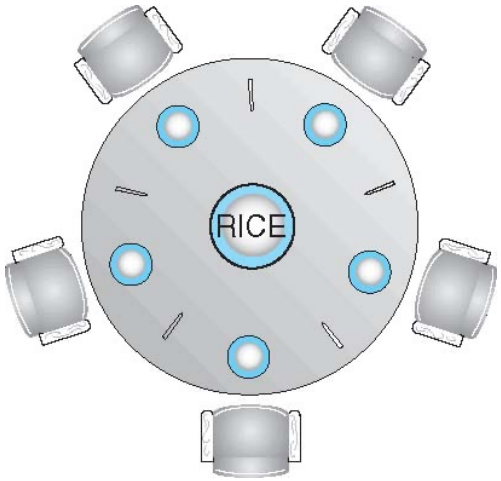
```
do {
        wait(rw_mutex);
          ...
        /* writing is performed */
          ...
        signal(rw_mutex);
} while (true);
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
    wait(mutex);
    read_count++; // Increase number of readers
    if (read_count == 1)
        wait(rw_mutex); // ensure no writer can enter while this current reader is inside critical section
    signal(mutex); //Other readers can enter while this reader is reading
                ...
     /* reading is performed */
                ...
    wait(mutex);
    read count--; // a reader want to leave
    if (read_count == 0)
        signal(rw_mutex); // write can enter
    signal(mutex); // reader leaves
}while (true);
```

Dining-Philosophers Problem



- Philosophers spend their lives alternating 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

chopstick[i]	1	1	1	1	1
i	0	1	2	3	4

- Although this solution guarantees that no two neighbors are eating simultaneously, it could still create a deadlock.
  - Suppose all five philosophers become hungry at same time and each grabs left chopstick. All elements of chopstick[5] will become 0

chopstick[i]	0	0	0	0	0
i	0	1	2	3	4

  - Each philosopher will be delayed forever to grab second chopstick on right.

Monitor Solution to Dining Philosophers

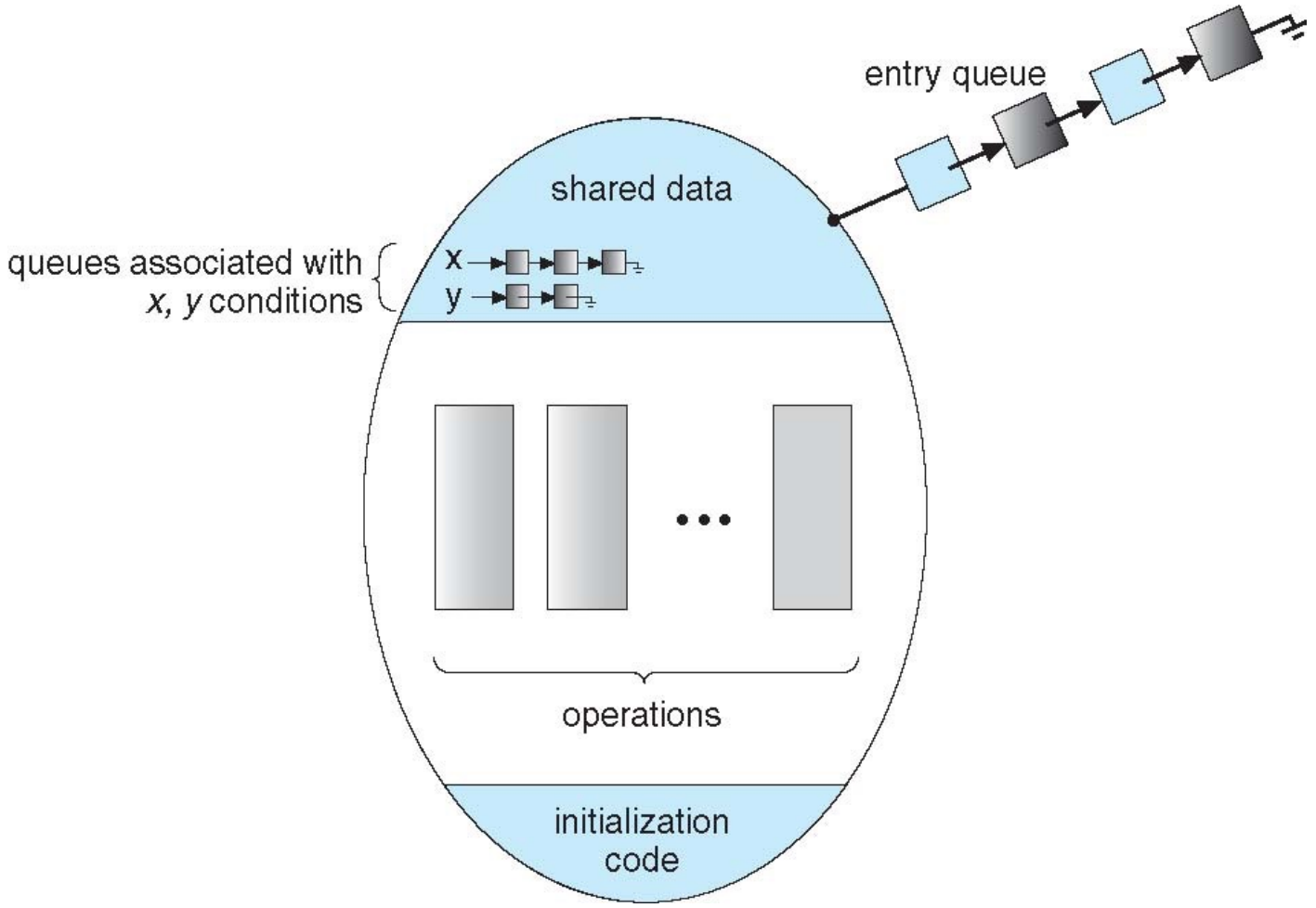
We need to distinguish among three states in which we may find a philosopher.

enum { **THINKING; HUNGRY, EATING**) state [5] ;

Philosopher i can set the variable state[i]= eating only if his two neighbors are not eating: state[(i+4)%5!=eating] and state[(i+1)%5!=eating]

We also need a variable condition self [5]; where philosopher i can delay himself when he is hungry but is unable to obtain the chopsticks he needs.

Recall: Monitor with Condition Variables



Monitor Solution to Dining Philosophers

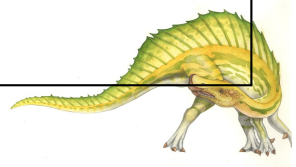
```
monitor DiningPhilosophers
{
    enum { THINKING; HUNGRY, EATING) state [5] ;
    condition self [5];

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

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

```
    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

End of Chapter 7