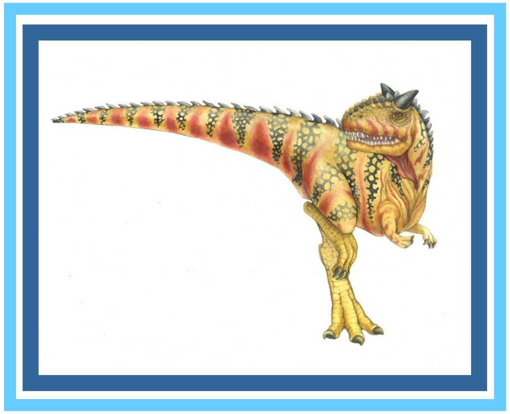
Synchronization Examples



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

- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0 (empty buffer)
- Semaphore **`empty`** initialized to the value n (no. of free slots)

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
while (true) {
      ...
      /* produce an item in next_produced */
      ...
    wait(empty); /*wait until there is an empty
             slot in the buffer*/
    wait(mutex); /*enter CS*/
       ...
      /* add next produced to the buffer */
       ...
    signal(mutex); /* leaves CS*/
    signal(full); /*signal that there is a new full
             slot an item availbale*/
}
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
while (true) {
    wait(full); /*wait until there is at least one
     full slot*/
    wait(mutex); /* enter CS*/
        ...
     /* remove an item from buffer to next_consumed */
        ...
    signal(mutex); /*leave CS*/
    signal(empty); /*Signal that one empty slot is
     free now*/
        ...
      /* consume the item in next consumed */
        ...
}
```

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
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities

Readers-Writers Problem (Cont.)

- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0

Readers-Writers Problem (Cont.)

- The structure of a writer process

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

Readers-Writers Problem (Cont.)

- The structure of a reader process

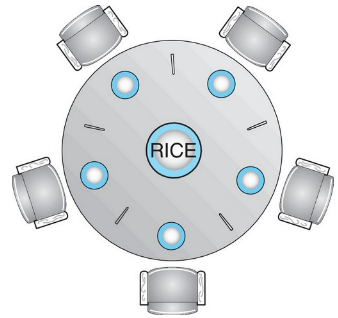
```
while (true){
            wait(mutex);
            read_count++;
            if (read_count == 1) /* first reader */
                  wait(rw_mutex);
                  signal(mutex);
         ...
            /* reading is performed */
         ...
            wait(mutex);
            read count--;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

Readers-Writers Problem Variations

- The solution in previous slide can result in a situation where a writer process never writes. It is referred to as the "First reader-writer" problem. (no reader kept waiting unless writer has permission to use shared object)
- The "Second reader-writer" problem is a variation the first reader-writer problem that state:
  - Once a writer is ready to write, no "newly arrived reader" is allowed to read.
- Both the first and second may result in starvation. leading to even more variations
- Problem is solved on some systems by kernel providing r**eader-writer locks**

Dining-Philosophers Problem

- N philosophers' sit at a round table with a bowel of rice in the middle.



- They spend their lives alternating thinking and eating.
- They do not 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, the shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm

- Semaphore Solution
- The structure of Philosopher *i*:

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

     /* eat for awhile */

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

     /* think for awhile */

}
```

- What is the problem with this algorithm?

Monitor Solution to Dining Philosophers

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

   void pickup (int i) {
          state[i] = HUNGRY;//philosopher is hungry//
          test(i); //try to get both chopsticks//
          if (state[i] != EATING)
                 self[i].wait; //wait if not allowed to eat//
   }

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

   }
```

Solution to Dining Philosophers (Cont.)

```
    void test (int i) {
            if ((state[(i + 4) % 5] != EATING) && //left not eating//
            (state[i] == HUNGRY) &&  //I is hungry//
            (state[(i + 1) % 5] != EATING) ) { //right not eating//
                 state[i] = EATING ; //allow philosopher i to eat//
               self[i].signal () ; // wake up I if waiting//
            }
    }

        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