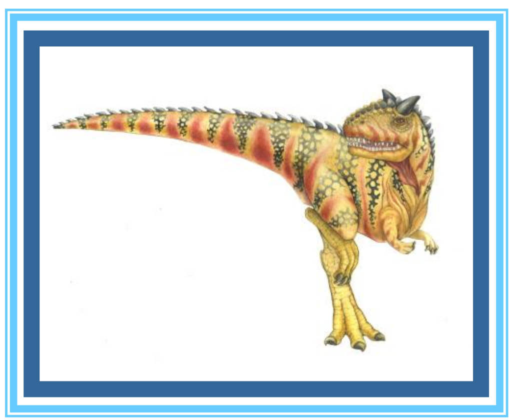
Chapter 7: Synchronization Examples



Classical Problems of Synchronization

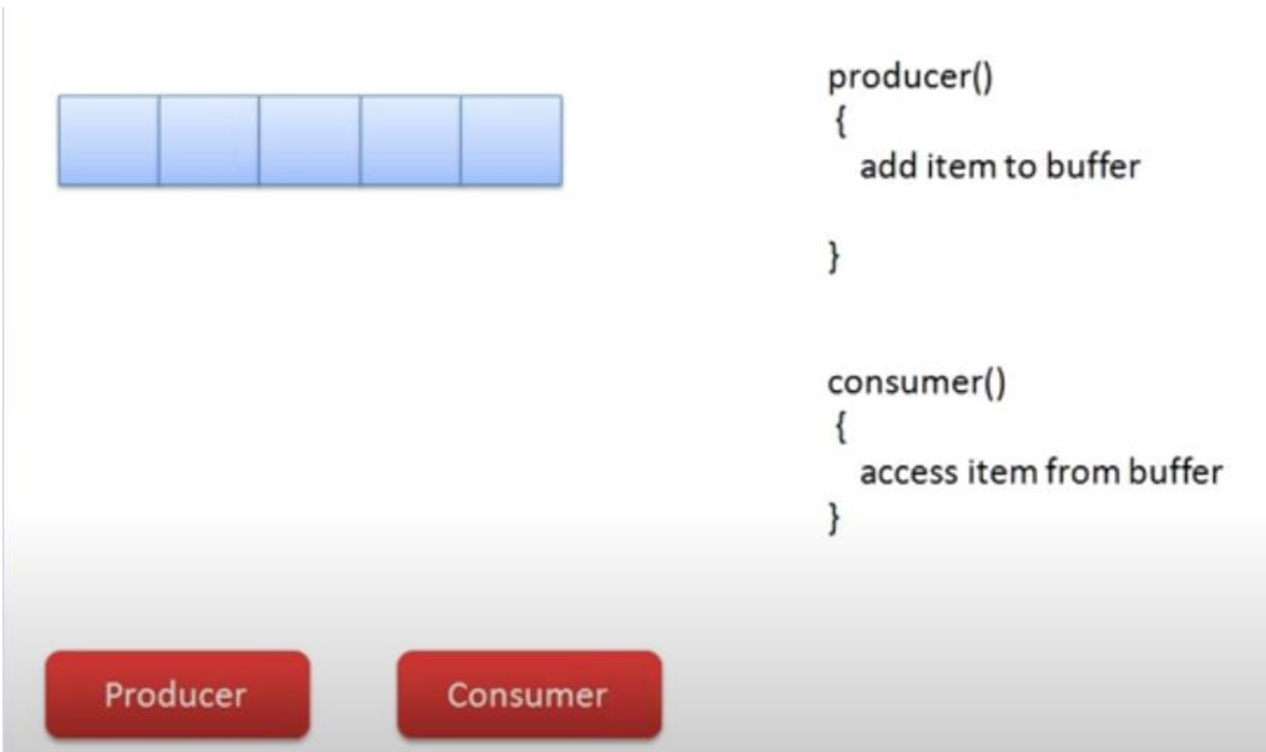
- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- *n* buffers, each can hold one item

semaphore locks used to control the buffer index

- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value n



Mutex=1, full=0, empty=n

- 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);
```

- 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);
```

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
- 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

```
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);
```

Readers and Writers Problem

Writer.
Will read data and modify it.

Only one writer is allowed to write at a time.

At time of writing, reading is not allowed.

Reader.
Will read only.

Multiple readers can read simultaneously.

At time of reading, writing is not allowed.

wait (mutex)
Read here
signal (mutex)

wait (mutex)
Read here
signal (mutex)

wait (mutex)
Read here
signal (mutex)

The last reader will release the mutex (or release the lock!)

```
wait (wrt)
Write here
signal (wrt)
```

readcnt = 0

wrt = 1

mutex = 1

```
wait(mutex)
readcount ++ ;
if (readcount == 1)
        wait (wrt) ;
signal(mutex)
// reading is performed

wait(mutex)
readcount  - - ;
if (readcount  == 0)
        signal (wrt) ;
signal(mutex)
```

Reader 1 wants to read.

```
wait (wrt)
Write here
signal (wrt)
```

readcnt = 1 | wrt = 0 | mutex = 0

```
wait(mutex)
readcount ++ ;
if (readcount == 1)
        wait (wrt) ;
signal(mutex)
// reading is performed

wait(mutex)
readcount - - ;
if (readcount  == 0)
        signal (wrt) ;
signal(mutex)
```

Reader 1 wants to read.

```
wait (wrt)
Write here
signal (wrt)
```

readcnt = 2 | wrt = 0 | mutex = 0

```
wait(mutex)
readcount ++ ;
if (readcount == 1)
        wait (wrt) ;
signal(mutex)
// reading is performed

wait(mutex)
readcount  - - ;
if (readcount  == 0)
        signal (wrt) ;
signal(mutex)
```

Reader 1 is reading.

Reader 2 wants to read.

False! Reader 1 has already informed the writer that it wants to read.

```
wait (wrt)
Write here
signal (wrt)
```

readcnt = 1 | wrt = 0 | mutex = 0

```
wait(mutex)
readcount ++ ;
if (readcount == 1)
        wait (wrt) ;
signal(mutex)
// reading is performed

wait(mutex)
readcount  - - ;
if (readcount  == 0)
        signal (wrt) ;
signal(mutex)
```

Reader 1 is reading.

Reader 2 is reading.

Reader1 wants to stop reading.

False! Reader 2 is still reading so writer cannot enter yet.

```
wait (wrt)
Write here
signal (wrt)
```

readcnt = 0 | wrt = -1 | mutex = 0

```
wait(mutex)
readcount ++ ;
if (readcount == 1)
        wait (wrt) ;
signal(mutex)
// reading is performed

wait(mutex)
readcount  - - ;
if (readcount  == 0)
        signal (wrt) ;
signal(mutex)
```

Reader 1 is reading.

Reader 2 is reading.

Reader1 wants to stop reading.

Writer wants to write.

Reader2 wants to stop reading.

```
wait (wrt)
Write here
signal (wrt)
```

readcnt = 0

wrt = 0

mutex = 0

```
wait(mutex)
readcount ++ ;
if (readcount == 1)
        wait (wrt) ;
signal(mutex)
// reading is performed

wait(mutex)
readcount  - - ;
if (readcount  == 0)
        signal (wrt) ;
signal(mutex)
```

Reader 1 is reading.

Reader 2 is reading.

Reader1 wants to stop reading.

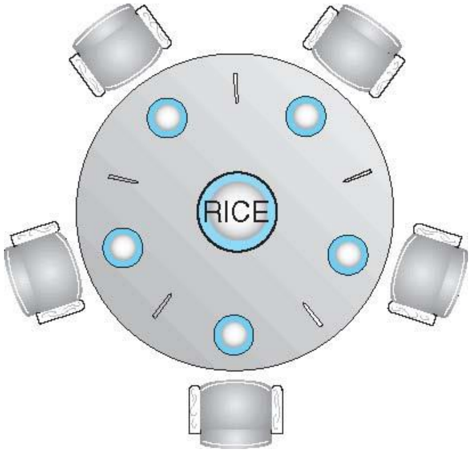
Writer wants to write.

No more readers left.

Readers-Writers Problem Variations

- ***First*** variation – no reader kept waiting unless writer has 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
  - writers might starve if there are always new readers
  - readers might starve if writers keep arriving
- Problem is solved on some systems by kernel providing reader-writer locks:
  - The kernel ensures fairness by preventing new readers from starting while a writer is waiting, and after the writer finishes, it allows waiting readers to read before another writer can proceed.

Dining-Philosophers Problem



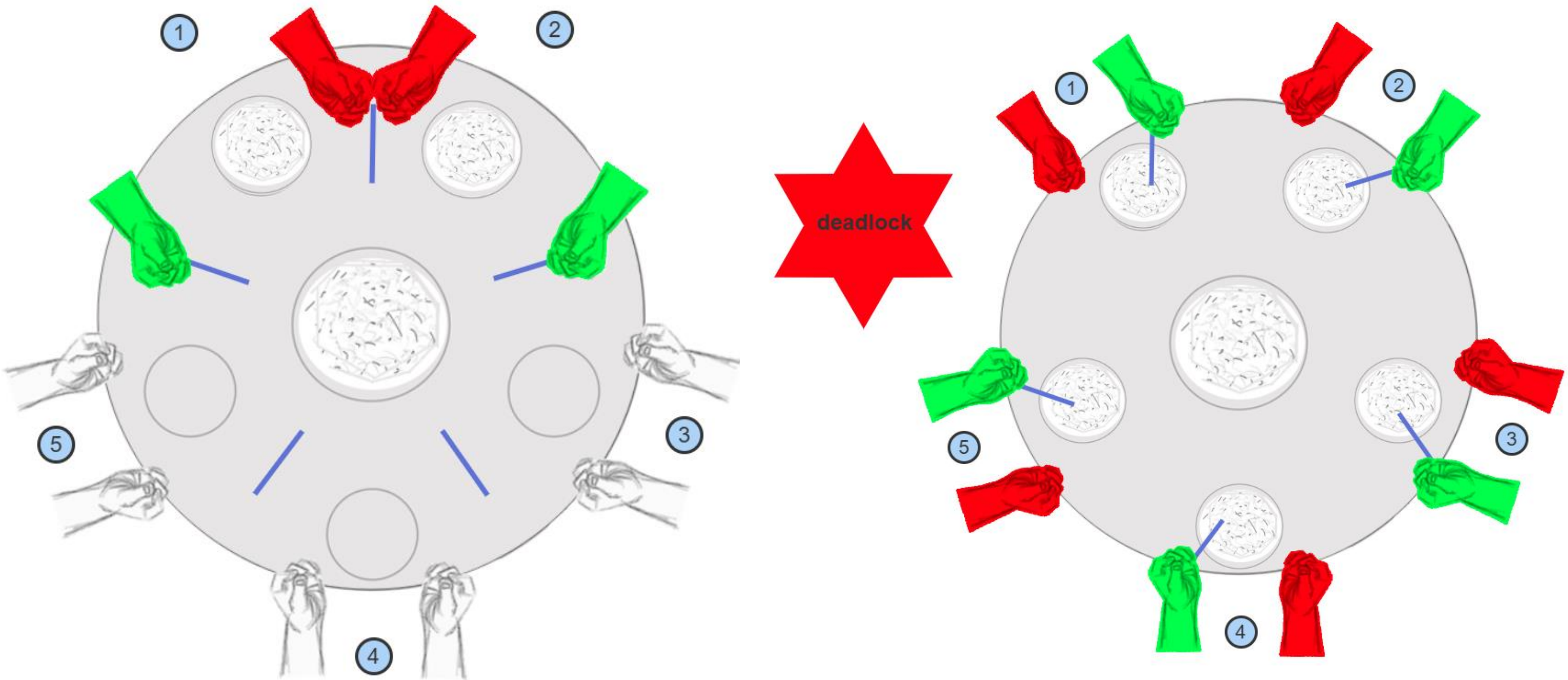
- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data 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?


1
2
3
4
5
deadlock
1
2
3
4
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 left and right neighbors
           test((i + 4) % 5);
           test((i + 1) % 5);
}
```

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
  - The system does not fairly allocate processes, prioritizing some philosophers over others.
  - If a philosopher on the left side of the table always manages to grab the forks before the philosopher on the right, the philosopher on the right might wait indefinitely and starve while others keep eating.

Synchronization Examples

- Solaris, Windows, Linux, and Pthreads all offer common synchronization mechanisms such as mutexes, semaphores, condition variables, and reader/writer locks, but the specific implementations vary.
- Windows has unique features like Slim Reader/Writer (SRW) Locks for fast synchronization.
- Linux offers futexes, which allow efficient synchronization between threads in user space without requiring full kernel involvement.

Python Example Using Monitors with Semaphores

```
import threading
import time
# Shared resource
shared_data = 0
# Count of readers currently accessing the resource
reader_count = 0
# Semaphore to control access to reader_count
mutex = threading.Semaphore(1)
# Semaphore to control access to the shared resource
rw_semaphore = threading.Semaphore(1)
```

Reader

```python
# Reader function
def reader(reader_id):
    global reader_count
    while True:
        # Enter critical section to update reader_count
        mutex.acquire()
        reader_count += 1
        if reader_count == 1:
            # If this is the first reader, lock the resource for reading
            rw_semaphore.acquire()
        mutex.release()
        # Simulate reading from the shared resource
        print(f"Reader {reader_id} is reading: {shared_data}\n")
        time.sleep(1)
        # Exit critical section to update reader_count
        mutex.acquire()
        reader_count -= 1
        if reader_count == 0:
            # If this was the last reader, release the lock on the resource
            rw_semaphore.release()
        mutex.release()
        # Simulate some delay before the reader tries to read again
        time.sleep(1)
```

```python
# Writer function
def writer(writer_id):
    global shared_data
    while True:
        # Lock the resource for writing
        rw_semaphore.acquire()
        # Simulate writing to the shared resource
        shared_data += 1
        print(f"Writer {writer_id} is writing: {shared_data}\n")
        time.sleep(1)

        # Release the resource after writing
        rw_semaphore.release()
        # Simulate some delay before the writer tries to write again
        time.sleep(2)

# Create and start reader and writer threads
if __name__ == "__main__":
    # 3 readers
    readers = [threading.Thread(target=reader, args=(i,)) for i in range(3)]
    # 2 writers
    writers = [threading.Thread(target=writer, args=(i,)) for i in range(2)]

    # Start all reader and writer threads
    for t in readers + writers:
        t.start()

    # Join threads (keep main program running)
    for t in readers + writers:
        t.join()
```

End of Chapter 7