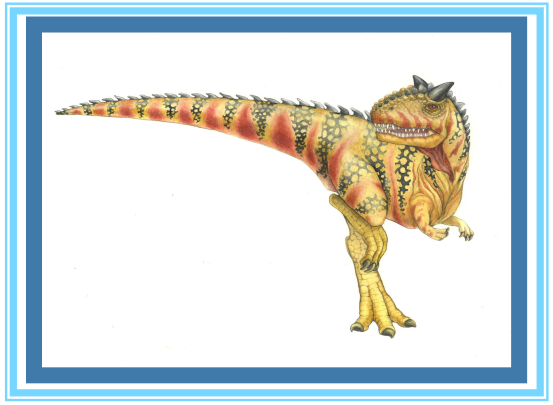
Chapter 6: Process Synchronization



Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations
  - wait (S) {
    while S <= 0
    ; // no-op
    S--;
    }
  - signal (S) {
    S++;
    }

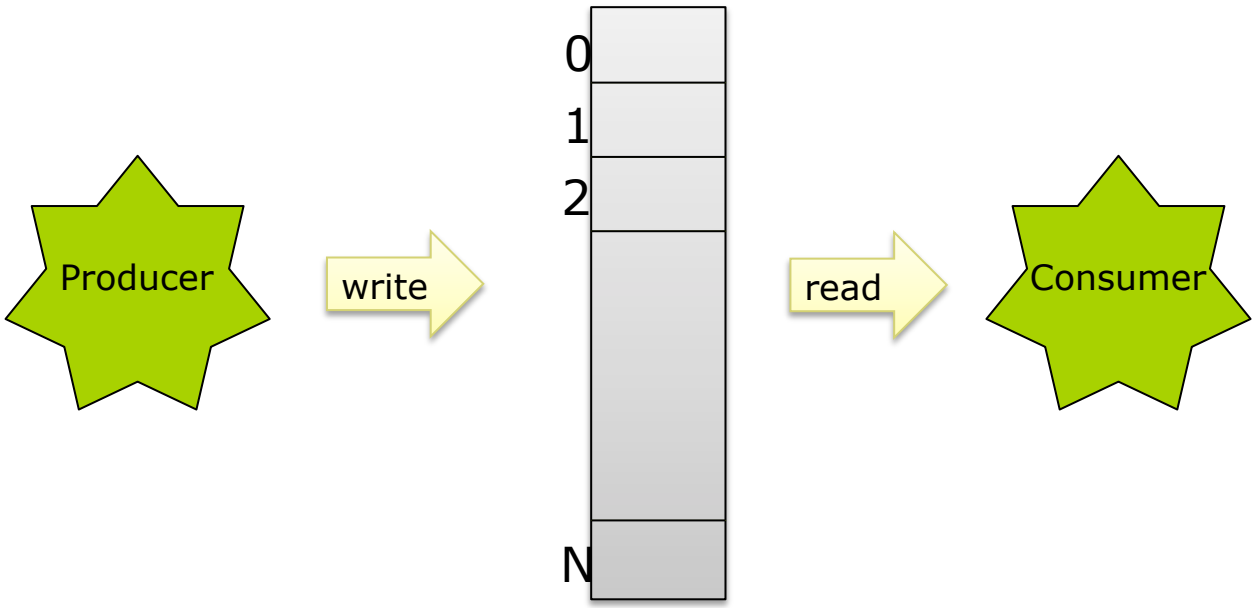
Semaphore as General Synchronization Tool

- **Counting** semaphore – integer value can range over an unrestricted domain
- **Binary** semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as **mutex locks**
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

```
Semaphore mutex;    //  initialized to 1
do {
    wait (mutex);
        // Critical Section
    signal (mutex);
        // remainder section
} while (TRUE);
```

Bounded Buffer Problem

- Producer cannot write when full, but wait until not full
- Consumer cannot read when empty, but wait until not empty
- Producer and Consumer cannot write/read simultaneously



BBP v3

Producer

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

Consumer

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

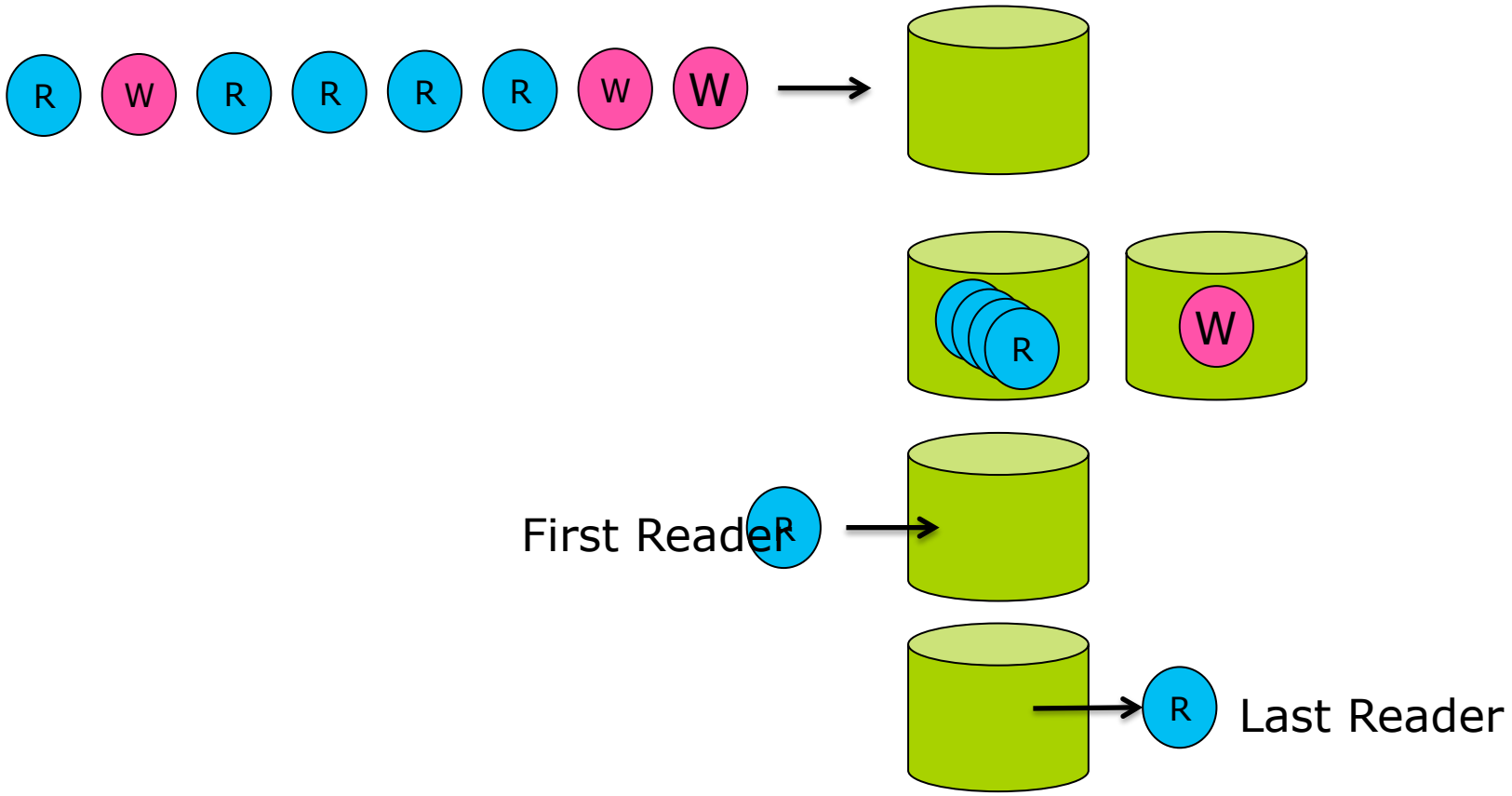
Classical Problems of Synchronization

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

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
- Concurrency
  - Allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time

First Reader-Writer problem



First Reader-Writer problem

- What are shared resources?
  - What shared variable is needed?
- Who are competing for which resources?
  - Binary semaphore for mutual exclusion
- How can we decide if I am the first reader or the last reader?

Writer

- Writer's algorithm
  - if another writer is writing or readers are reading (lock is held)
    - then wait
  - otherwise, lock and write
  - release lock and notify when done

Writer

- The structure of a writer process

```
do {
        wait (wrt) ;

            //    writing is performed

        signal (wrt) ;
} while (TRUE);
```

Reader

- Reader's algorithm
  - If I am the first reader
    - if a writer is writing, wait
    - or hold writing lock
  - read data
  - If I am the last reader
    - release writing lock

```
do {
        if (I am the first reader)
            wait (wrt) ;

        // read data

        if (I am the last reader)
            signal (wrt) ;

} while (TRUE);
```

Reader

- Decide if I am the first/last reader
- Count the number of readers in database
- Share the counting variable
- Mutex

Readers-Writers Problem (Cont.)

```
do {
        wait (mutex) ;
                readcount ++ ;
                        if (readcount == 1)
                                wait (wrt) ;
        signal (mutex)

                // Read data

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

} while (TRUE);
```

- //WRITER

```
do {
    wait (wrt) ;
    //    writing is performed
    signal (wrt) ;
} while (TRUE);
```

- Writer Starvation?
- No Writer Starvation code?

Readers/Writers/No starvation

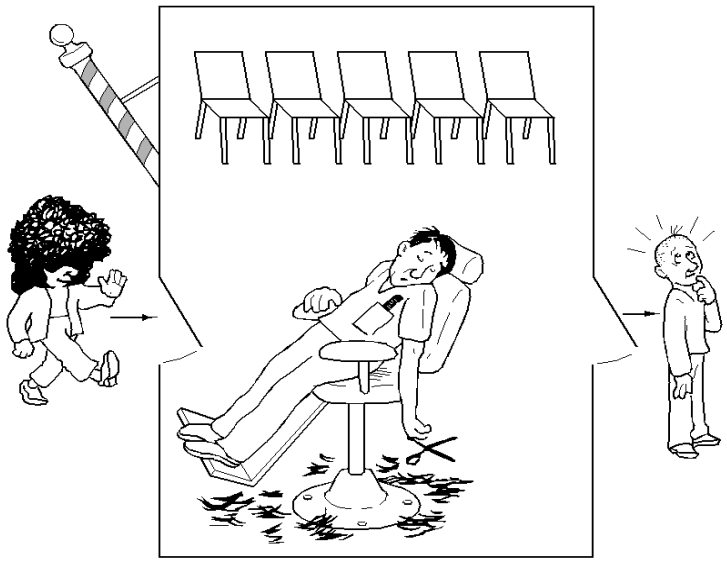
```
do {
        wait(r);
        wait (mutex) ;
                readcount ++ ;
                if (readcount == 1)
                        wait (wrt) ;
        signal (mutex)
        signal(r);
        // Read data
        wait (mutex) ;
                readcount  - - ;
                if (readcount  == 0)
                        signal (wrt) ;
        signal (mutex) ;

} while (TRUE);
```

- //WRITER

```
do {
    wait(r)
    wait (wrt) ;
     //    writing is performed
     signal (wrt) ;
     signal(r);
} while (TRUE);
```

Sleeping Barber Problem



- There is one barber, and n chairs for waiting customers
- If there are no customers, then the barber sits in his chair and sleeps
- When a new customer arrives and the barber is sleeping, then he will wakeup the barber
- When a new customer arrives, and the barber is busy, then he will sit on the chairs if there is any available, otherwise (when all the chairs are full) he will leave.

Barber Shop Hints

Consider the following:

- Customer threads should invoke a function named `getHairCut`.
- If a customer thread arrives when the shop is full, it can invoke `balk`, which exits.
- Barber threads should invoke `cutHair`.
- When the barber invokes `cutHair` there should be exactly one thread invoking `getHairCut` concurrently.

Sleeping Barber Solution

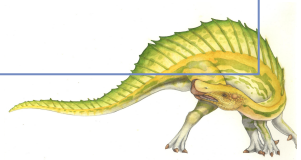
```
void customer (void){
        semWait(mutex);
        if (customers==n+1) {
                semSignal(mutex);
                balk();
        }
        customers +=1;
        semSignal(mutex);

        semSignal(customer);
        semWait(barber);
        getHairCut();

        semWait(mutex);
        customers -=1;
        semSignal(mutex);
}
```

```
void barber (void){
    semWait(customer);
    semSignal(barber);
    cutHair();
}
```

```
int customers = 0;
mutex = Semaphore(1);
customer = Semaphore(0);
barber = Semaphore(0);
```



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