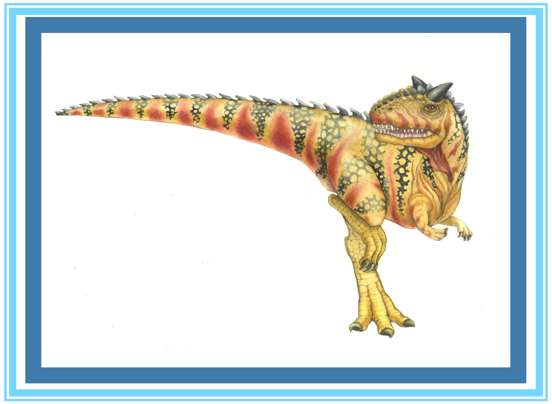
Chapter 6: Process Synchronization



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) {

```
wait (S) {
        while S <= 0
            ; // no-op
          S--;
   }
```

  - signal (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);
```

Implementation

- Continual Loop in entry code: waste of CPU resource
  - Busy-waiting semaphore: called *spinlock*
- How can we modify wait() and signal() without busy waiting?
- In wait(), when semaphore value is non-positive, block instead of busy-waiting
- blocking
  - adds the process into the waiting queue for the semaphore
  - and sets process state to *waiting*
- When signal() is called
  - wakeup() the blocked process in the waiting queue
  - wakeup() changes the process from *waiting* to *ready,* put in ready queue

Semaphore Implementation with no Busy waiting

- Semaphore data structure:
  - ```
    typedef struct {
      int value;
      struct process *list;
    } semaphore;
    ```
- Two system calls:
  - **block()** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup()** – remove one of processes in the waiting queue and place it in the ready queue


    ```

Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of wait:

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```

Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of signal:

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

Semaphore without busy-waiting

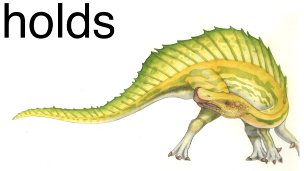
- Negative semaphore: # of processes waiting
- Semaphore waiting list
  - list of PCBs
  - FIFO queue for bounded-waiting, but…
- wait() and signal() are critical section

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

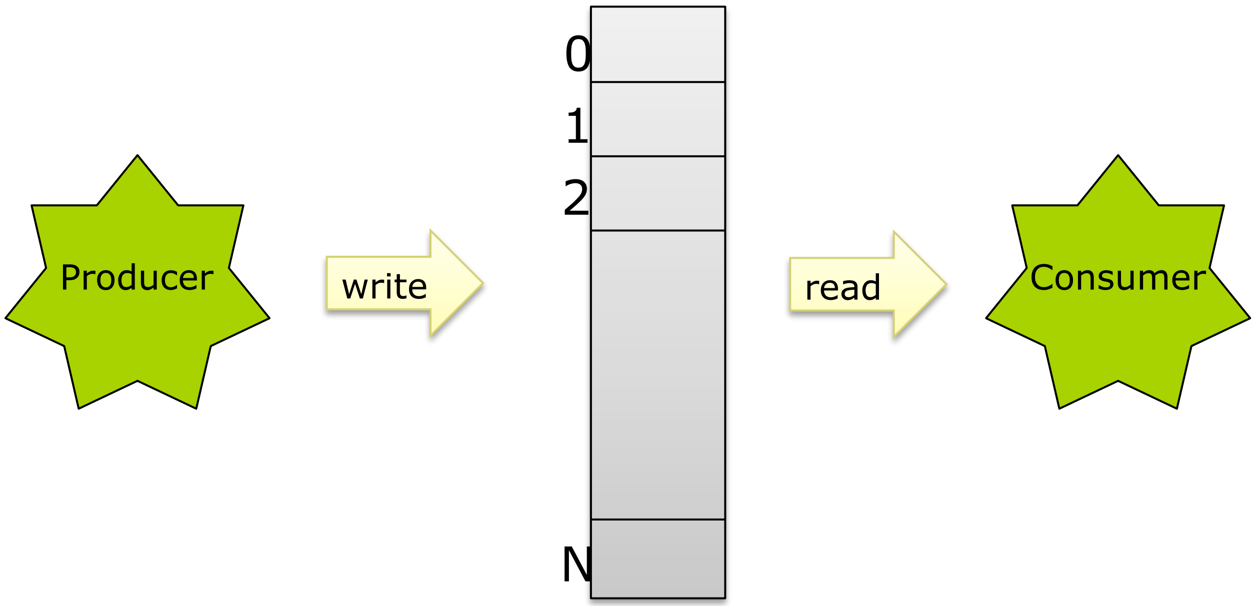
P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

- **Starvation** – indefinite blocking
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**



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 v1

Producer

```
do  {
    //   produce an item
    wait (empty);

    //  add the item to the  buffer

} while (TRUE);
```

Consumer

```
do {

    //  remove an item from  buffer

    signal (empty);
    //  consume the item
} while (TRUE);
```

Bounded-Buffer Problem

- *N* buffers, each can hold one item
- Semaphore mutex initialized to the value 1
- Semaphore full initialized to the value 0
- Semaphore empty initialized to the value N

BBP v2

Producer

```
do  {
    //   produce an item
    wait (empty);

    //  add the item to the  buffer

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

Consumer

```
do {
    wait (full);

    //  remove an item from  buffer

    signal (empty);
    //  consume the item
} while (TRUE);
```

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);
```

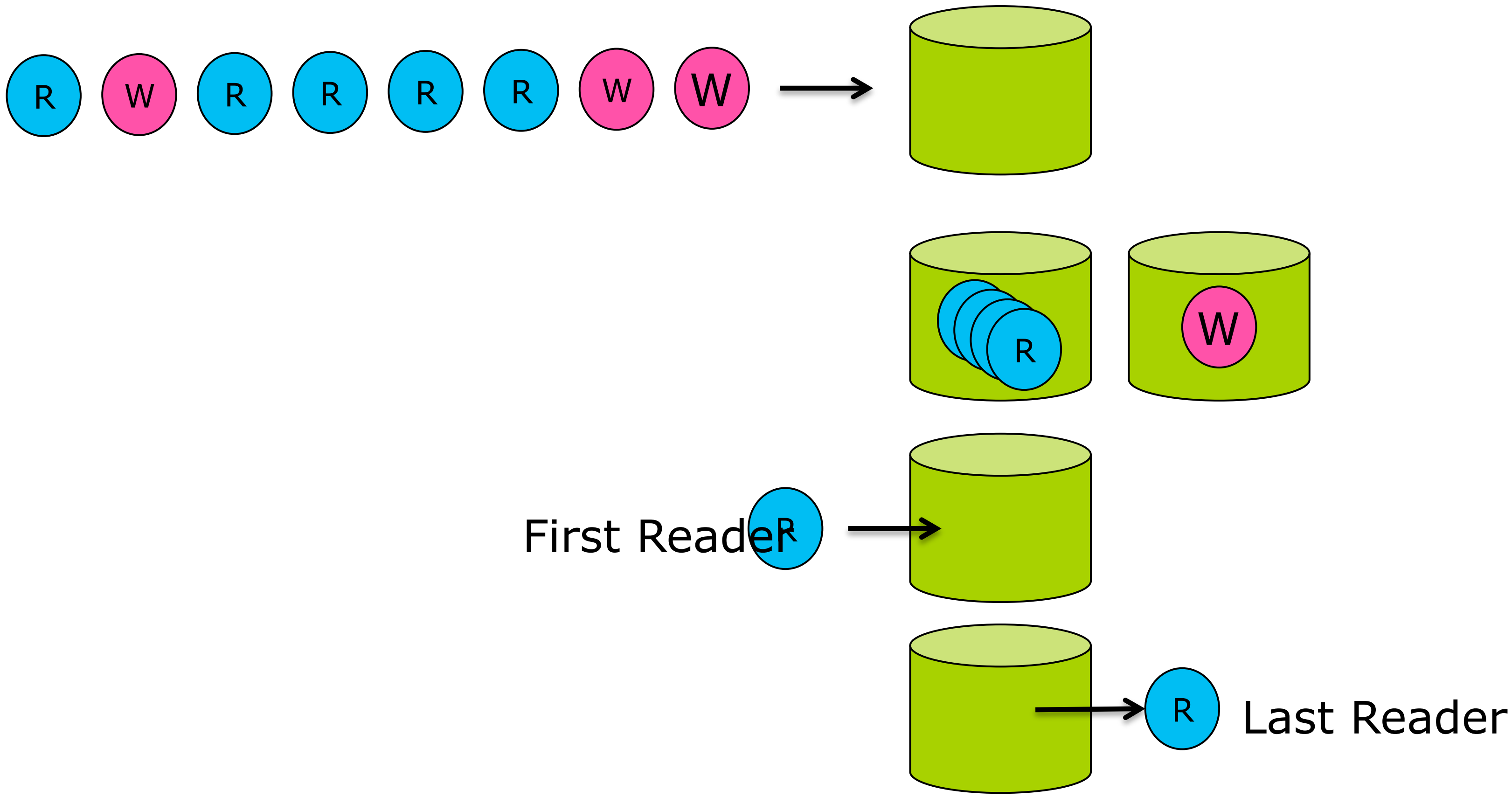
Classical Problems of Synchronization

- 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.)

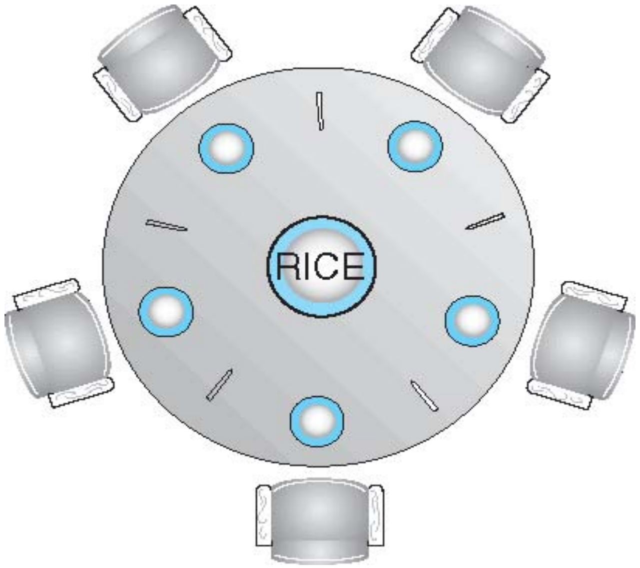
```
do {
        wait (mutex) ;
                readcount ++ ;
                        if (readcount == 1)
                                wait (wrt) ;
        signal (mutex)

                // Read data

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

} while (TRUE);
```

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