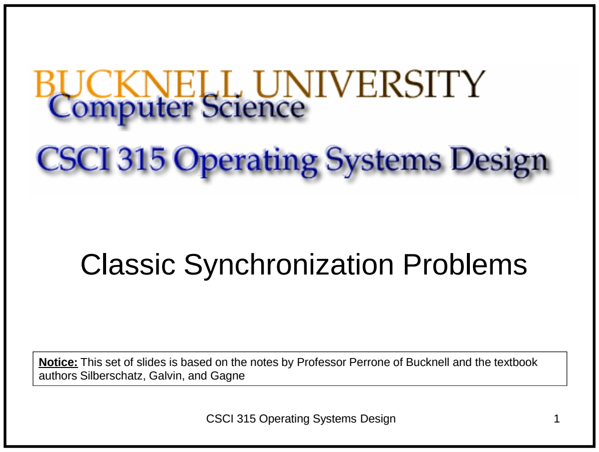
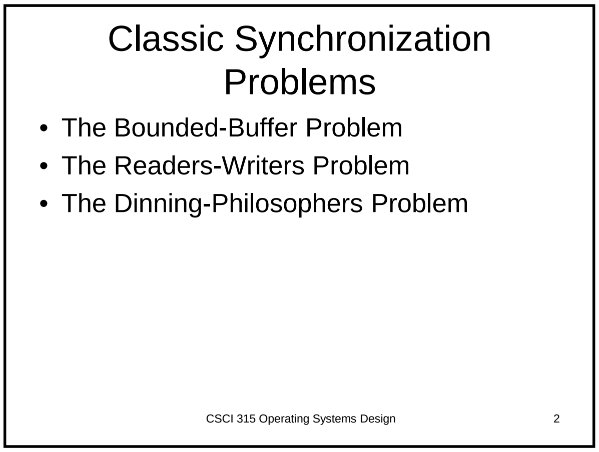


BUCKNELL UNIVERSITY
Computer Science

CSCI 315 Operating Systems Design

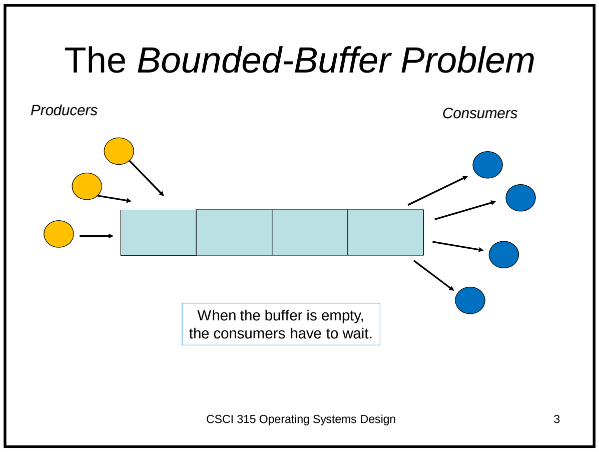
Classic Synchronization Problems

Notice: This set of slides is based on the notes by Professor Perrone of Bucknell and the textbook authors Silberschatz, Galvin, and Gagne



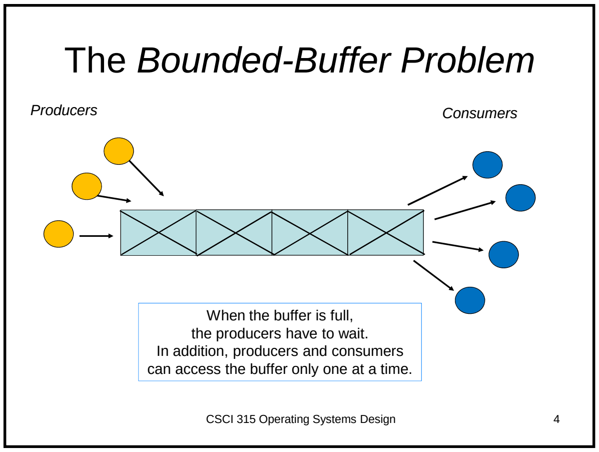
Classic Synchronization Problems

- The Bounded-Buffer Problem
- The Readers-Writers Problem
- The Dinning-Philosophers Problem



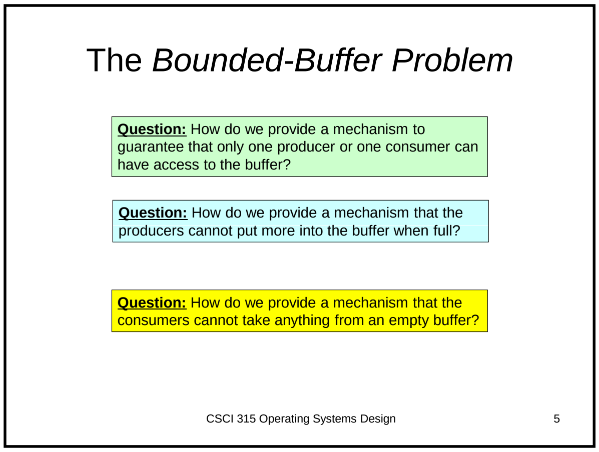
The *Bounded-Buffer Problem*

When the buffer is empty,
the consumers have to wait.



The *Bounded-Buffer Problem*

When the buffer is full,
the producers have to wait.
In addition, producers and consumers
can access the buffer only one at a time.

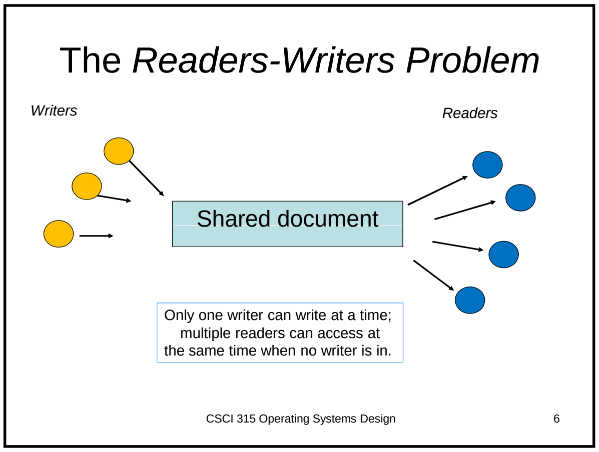


The *Bounded-Buffer Problem*

Question: How do we provide a mechanism to guarantee that only one producer or one consumer can have access to the buffer?

Question: How do we provide a mechanism that the producers cannot put more into the buffer when full?

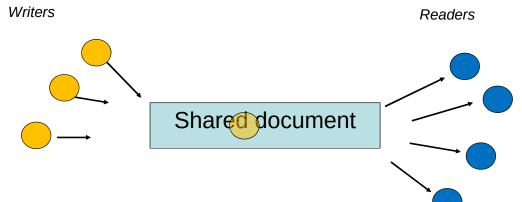
Question: How do we provide a mechanism that the consumers cannot take anything from an empty buffer?



The *Readers-Writers Problem*

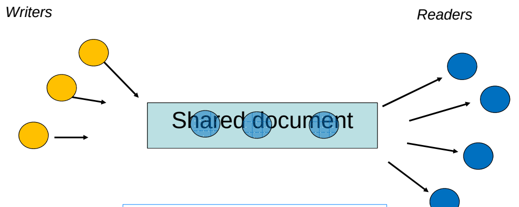
Only one writer can write at a time;
multiple readers can access at
the same time when no writer is in.

The *Readers-Writers Problem*



When one write is using the buffer,
no one else can access it.

The *Readers-Writers Problem*



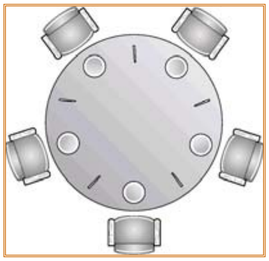
Multiple readers can access the buffer
at the same time,
no writers can when readers are in.

The *Readers-Writers Problem*

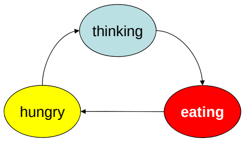
Question: How do we provide a mechanism to guarantee that only one writer can access to the buffer?

Question: How do we provide a mechanism that allows multiple readers access the buffer when no writer is in?

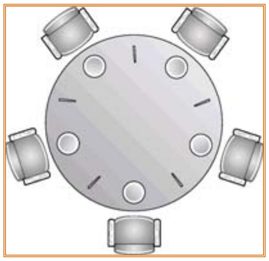
The *Dining-Philosophers* Problem



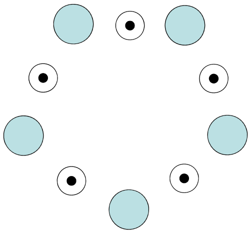
The *Dining-Philosophers* Problem

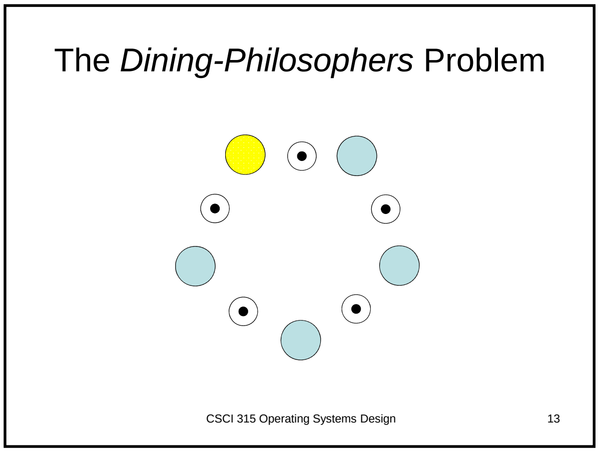


State diagram for a philosopher

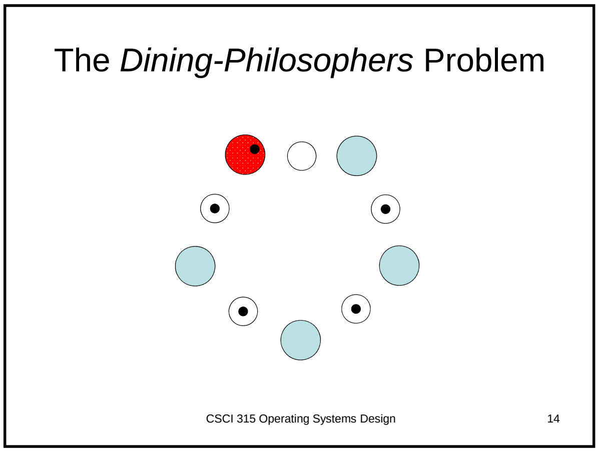


The *Dining-Philosophers* Problem

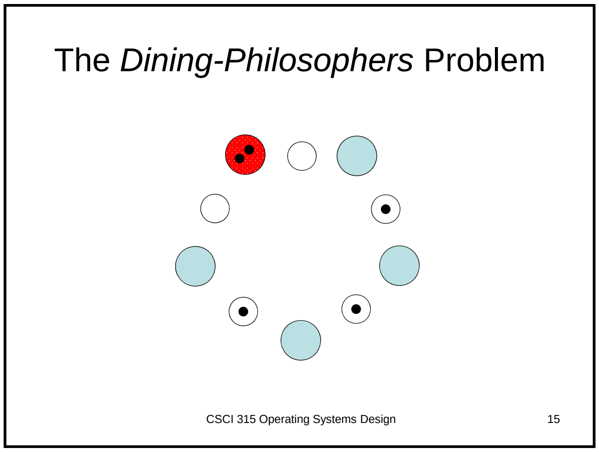




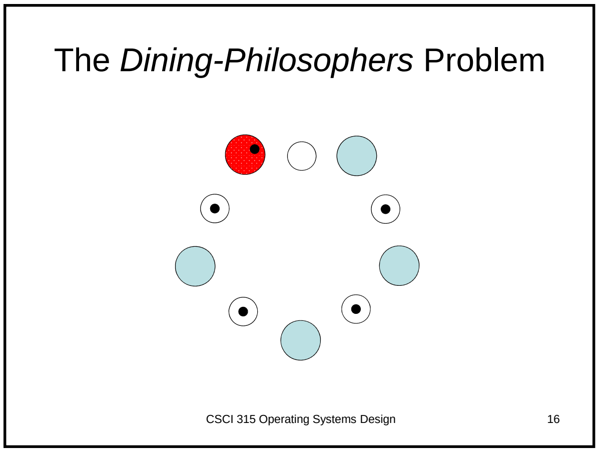
The *Dining-Philosophers* Problem



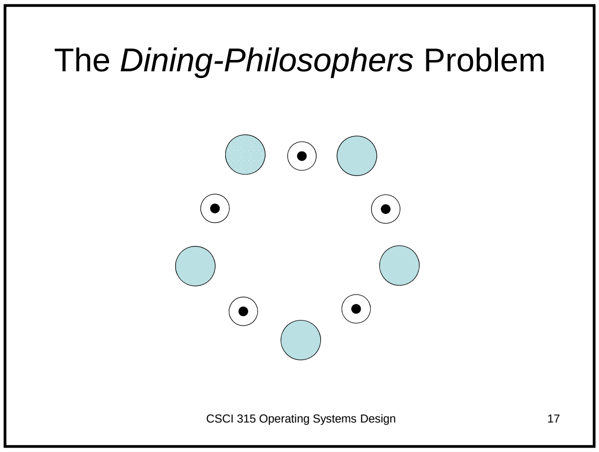
The *Dining-Philosophers* Problem



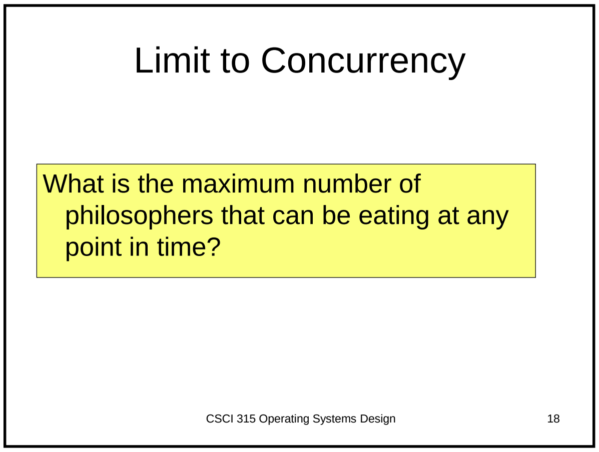
The *Dining-Philosophers* Problem



The *Dining-Philosophers* Problem



The *Dining-Philosophers* Problem



Limit to Concurrency

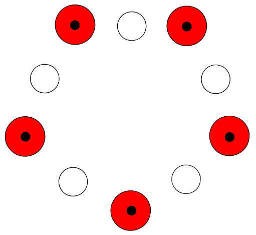
What is the maximum number of philosophers that can be eating at any point in time?

Philosopher's Behavior

- Grab chopstick on left
- Grab chopstick on right
- Eat
- Put down chopstick on right
- Put down chopstick on left

How well does this work?

The *Dining-Philosophers* Problem



The *Dining-Philosophers* Problem

Question: How many philosophers can eat at once? How can we generalize this answer for *n* philosophers and *n* chopsticks?

Question: What happens if the programmer initializes the semaphores incorrectly? (Say, two semaphores start out a zero instead of one.)

Question: How can we formulate a solution to the problem so that there is no deadlock or starvation?

Overall Questions

Question: What are the similarities among these problems?

Question: What are the differences?

Make sure to read the relevant sections of the textbook and understand the solutions there.

A Closer Look at the Readers/Writers Problem

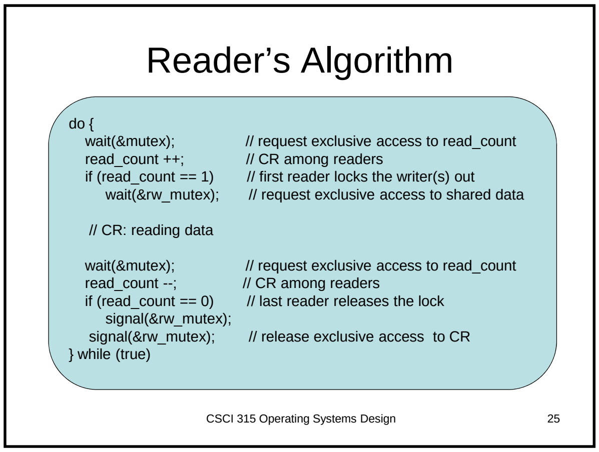
- From a writer's point of view, exclusive access is needed. That is at any moment, only one writer is allowed (no other writers, no readers) in the CR.
- From a reader's point of view, if a writer is in the CR, no readers can access the CR; if a reader (or no one) is in CR, many readers can get into the CR.

Writer's Algorithm

```
do {
    wait(&rw_mutex);          // request exclusive access

     // CR: writing data

     signal(&rw_mutex);       // release exclusive access
} while (true)
```



Reader’s Algorithm

```
do {
    wait(&mutex);                  // request exclusive access to read_count
    read_count ++;                 // CR among readers
    if (read_count == 1)           // first reader locks the writer(s) out
         wait(&rw_mutex);          // request exclusive access to shared data

     // CR: reading data

    wait(&mutex);                  // request exclusive access to read_count
    read_count --;                // CR among readers
    if (read_count == 0)           // last reader releases the lock
         signal(&rw_mutex);
     signal(&rw_mutex);            // release exclusive access  to CR
} while (true)
```