CSCI315 – Operating Systems Design

Department of Computer Science
Bucknell University

Synchronization Example: Dinning Philosophers

Ch 6.6

This set of notes is based on notes from the textbook authors, as well as L. Felipe Perrone, Joshua Stough, and other instructors.
Xiannong Meng, Fall 2021.

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. The processes P_0 and P_1 are likely ***deadlocked***.

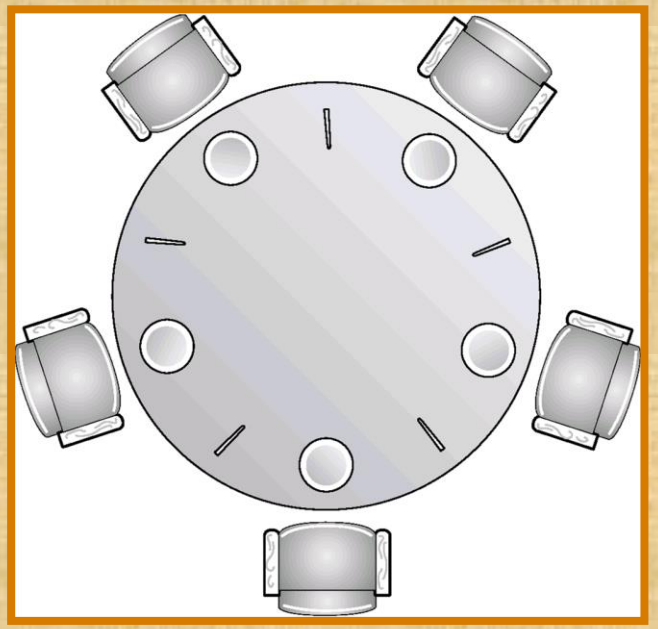
P_0	P_1
acquire(S);	**acquire(Q);**
acquire(Q);	**acquire(S);**
.	.
.	.
.	.
release(S);	**release(Q);**
release(Q);	**release(S);**

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended. E.g., the process in the CR goes into an infinite loop …

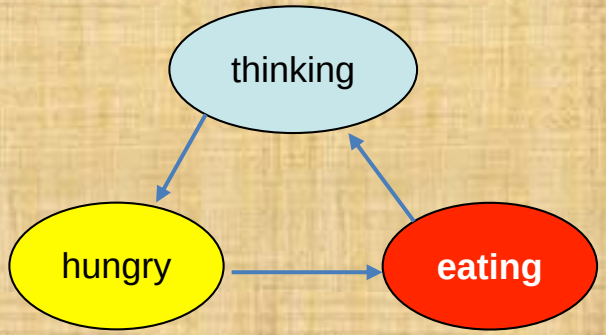
The Dining-Philosophers Problem

N philosophers sit with N chopsticks. Anyone who wants to eat will need both chopsticks. They can only grab one chopstick at a time.

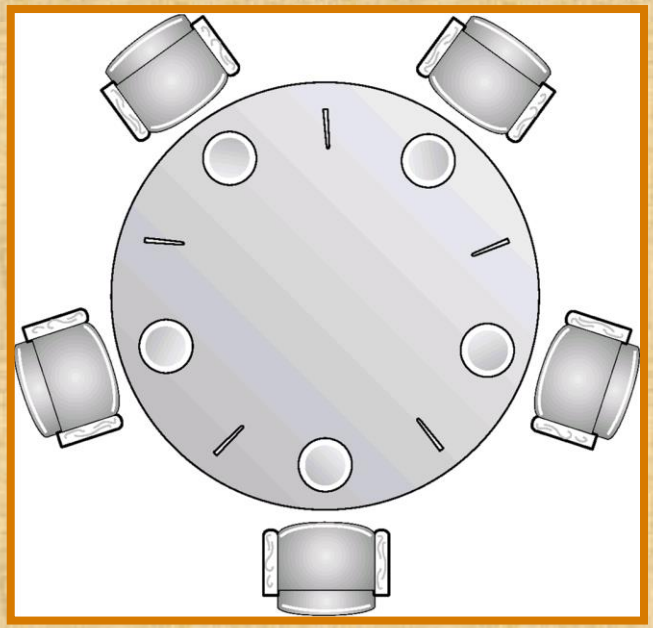
If one is able to get both chopsticks, they will eat. Otherwise, they wait for the chopsticks, or they are in a thinking state (idle).



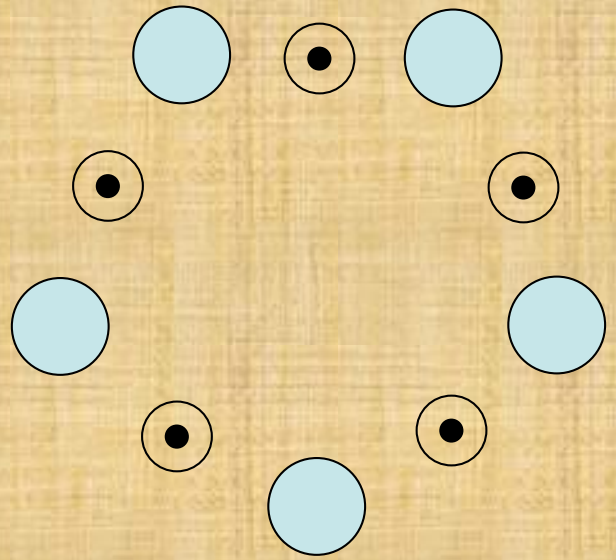
The Dining-Philosophers Problem



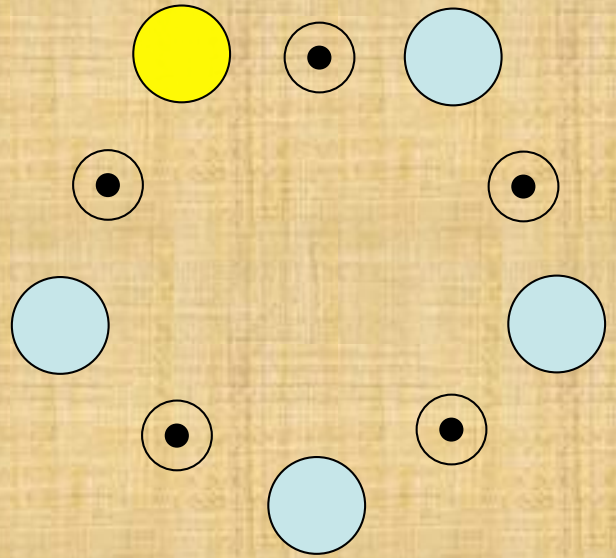
State diagram for a philosopher. We will use the colors to indicate the state of a philosopher.



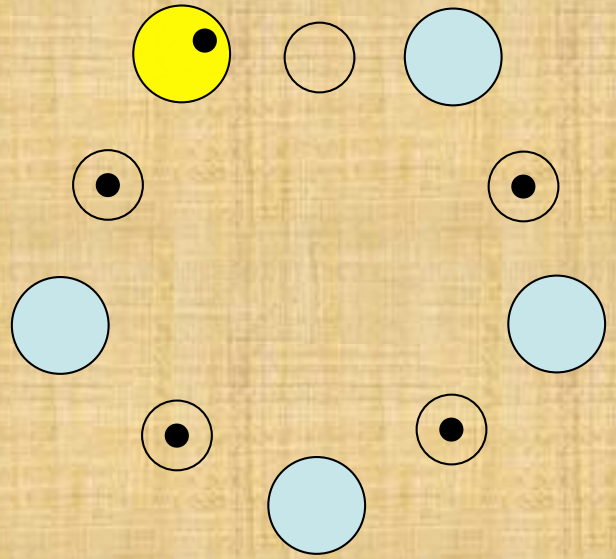
The Dining-Philosophers Problem



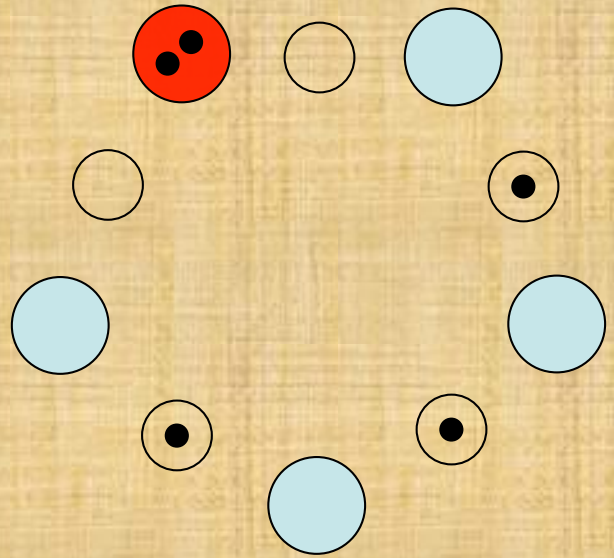
The Dining-Philosophers Problem



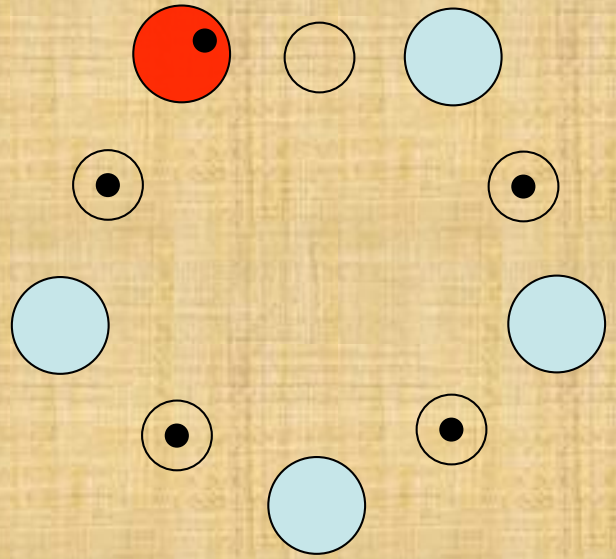
The Dining-Philosophers Problem



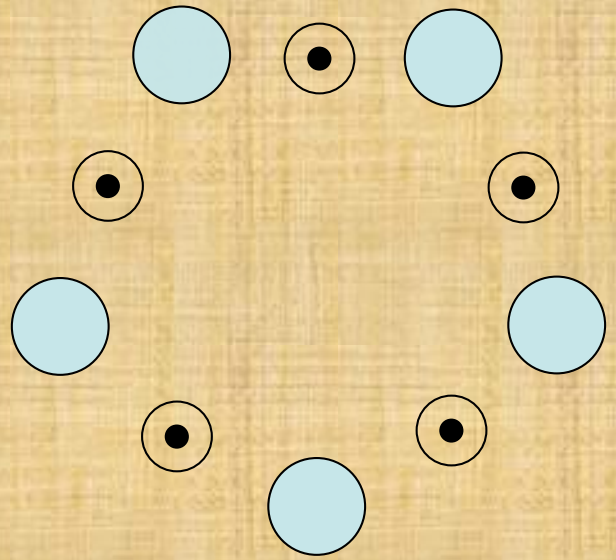
The Dining-Philosophers Problem



The Dining-Philosophers Problem



The Dining-Philosophers Problem



Limit to Concurrency

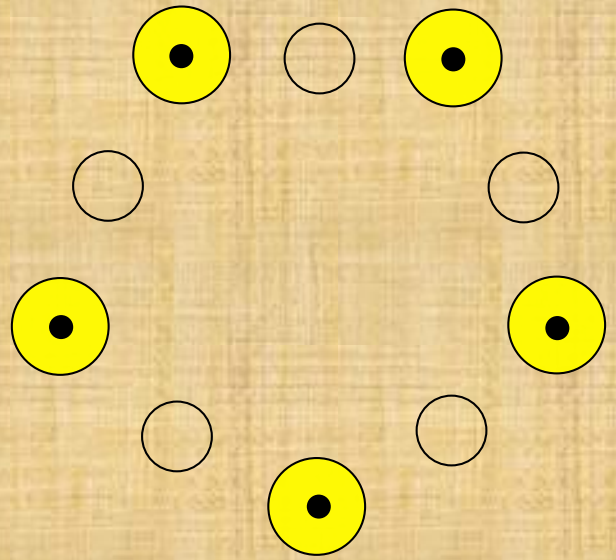
What is the maximum number of philosophers that can be eating at any point in time?

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?

Monitor

- Semaphores are low-level synchronization resources.
- A programmer's honest mistake can compromise the entire system (well, that is almost always true). We should want a solution that reduces the risk.
- The monitor is one such data type:

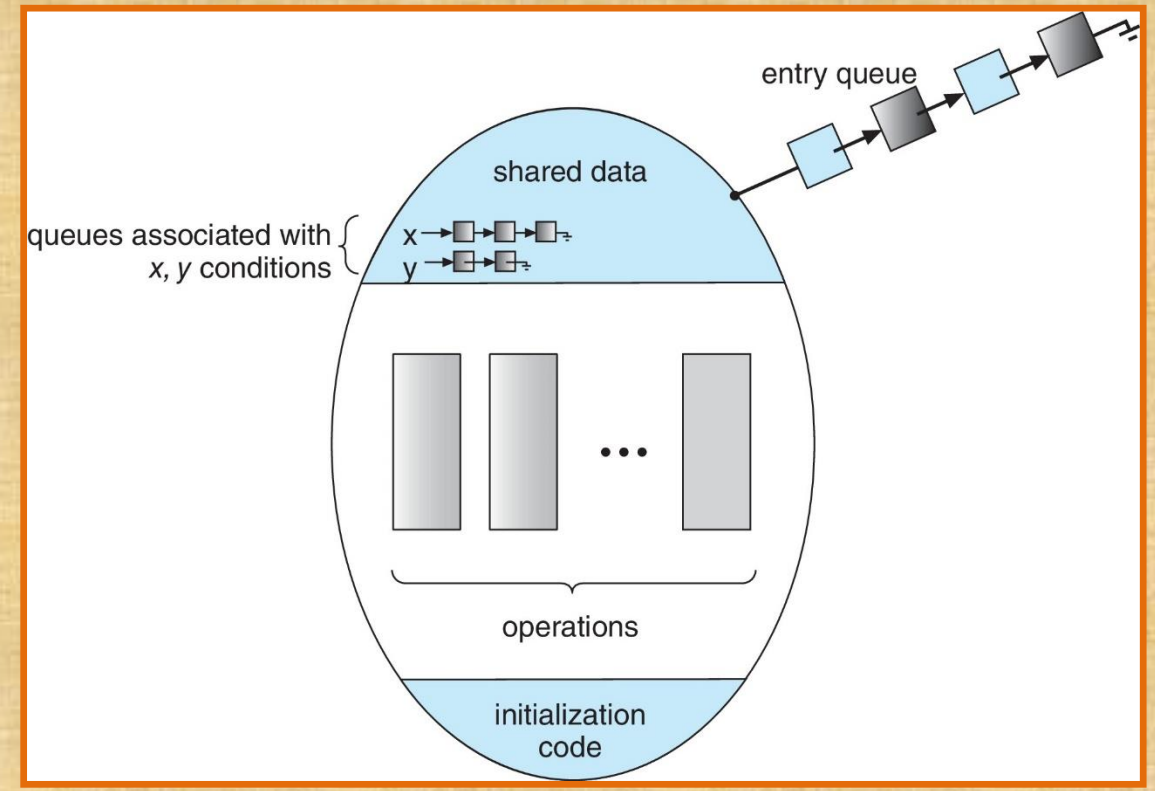
```
monitor mName {
  // declare shared variables
  procedure P1 (…) {
     …
  }
  procedure Pn (…) {
     …
  }
  init code (…) {
    ….
  }
}
```

A *procedure* can access only local variables defined within the monitor.

There cannot be concurrent access to procedures within the monitor (only one process/thread can be *active* in the monitor at any given time).

Condition variables: queues are associated with variables. Primitives for synchronization are wait and signal.

Monitor


entry queue
shared data
queues associated with
x, y conditions
x
y
operations
initialization
code

Conditional Variables

- Monitors provide a high level structure that the programmers don’t need to worry about the details.
- However a monitor contains many pieces, e.g., procedures and variables, the monitor implementations do have to take care of the synchronizations.

Conditional Variables

- Conditional variables are a structure inside the monitor to provide such a mechanism.
- The only two operations allowed on a conditional variables is wait() and signal().
- They look similar to the operations on semaphore. But usually “signal-and-continue” is a better option in monitor.
- We can use semaphore to implement a monitor.

Fig 7.7 - monitor solution

```
monitor DiningPhilosophers {
   enum {THINKING, HUNGRY, EATING} state [5] ;
   condition self [5];  // conditional variable

   void pickup (int i) {
      state[i] = HUNGRY;
      test(i);  // see test() on next page
      if (state[i] != EATING)
         self[i].wait();
   }  // end of pickup()

   void putdown (int i) {
      state[i] = THINKING;
      test((i + 4) % 5);     // signal left and right
      test((i + 1) % 5);
    }  // end of putdown()
```

```
   void test (int i) {
      if ((state[(i + 4) % 5] != EATING) &&
          (state[i] == HUNGRY) &&
          (state[(i + 1) % 5] != EATING) ) {
                state[i] = EATING ;
                self[i].signal () ;
      }
    }   // end of test()

    void initialization_code() {
       for (int i = 0; i < 5; i++)
          state[i] = THINKING;
      }   // end of initialization()
}   // end of monitor()
```

The Testing Logic for the Monitor Solution

```
#define NUM_PHIL 5  // some random ints

monitor DiningPhilosophers people[NUM_PHIL];

for (i = 0; i < NUM_PHIL; i ++) {
   pthread_create(tid[i], NULL, phil_routine, &i);
 }

void *phil_routine(void *argv) {
    int i = (*(int *)argv);
    do {
        people.pickup(i);
        …
        eat(i);
        …
        people.putdown(i);
        // some delay
    } while (1);
}
```

Sleeping Barber

https://images.app.goo.gl/HDrPq9ePwpJPS2NM7