Operating System Design

Processes Synchronization

Neda Nasiriani
Fall 2018

Process Synchronization

Condition Variables

- Yet another synchronization tool
- If we want to check a if a condition holds or not before continuing the execution (parent process checking if the child process is done)
- condition variable is an explicit queue
  - Threads can put themselves on the queue when the condition does not hold (by invoking wait on the condition)
  - Some other thread, when it changes the condition, can then wake one (or more) of those waiting threads and thus allow them to continue (by invoking signal on the condition).
- Difference from semaphores?
  - Does not keep a count but only put processes into sleep or wake them based on the state of the condition

Semaphores

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`**

```
wait(S) {
    while (S <= 0)
       ; // busy wait
    S--;
}
```

```
signal(S) {
    S++;
}
```

Atomically

Can we avoid busy waiting?!?

```
acquire() {
    while (!available)
       ; /* busy wait */
    available = false;;
}
```

```
release() {
   available = true;
}
```

```
wait(S) {
    while (S <= 0)
       ; // busy wait
    S--;
}
```

```
signal(S) {
    S++;
}
```

```
do {
  waiting[i] = true;
  key = true;
  while (waiting[i] && key)
    key = test_and_set(&lock);
  waiting[i] = false;

    /* critical section */

  j = (i + 1) % n;
  while ((j != i) && !waiting[j])
    j = (j + 1) % n;

  if (j == i)
    lock = false;
  ...
    ...alse;

    ...ection */
```

```
do {
  while (compare_and_swap(&lock, 0, 1) != 0)
    ; /* do nothing */

    /* critical section */

  lock = 0;

    /* remainder section */
} while (true);
```

```
do {
  while (test_and_set(&lock))
    ; /* do nothing */

    /* critical section */

  lock = false;

    /* remainder section */
} while (true);
```

No Busy Waiting!

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list

```
typedef struct {
        int value;
        struct process *list;
} semaphore;
```

- Two operations:
  - **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

No Busy Waiting!

- Two operations:
  - **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

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

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

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 risk.
- The solution can take the shape of high-level language constructs, as the monitor type:

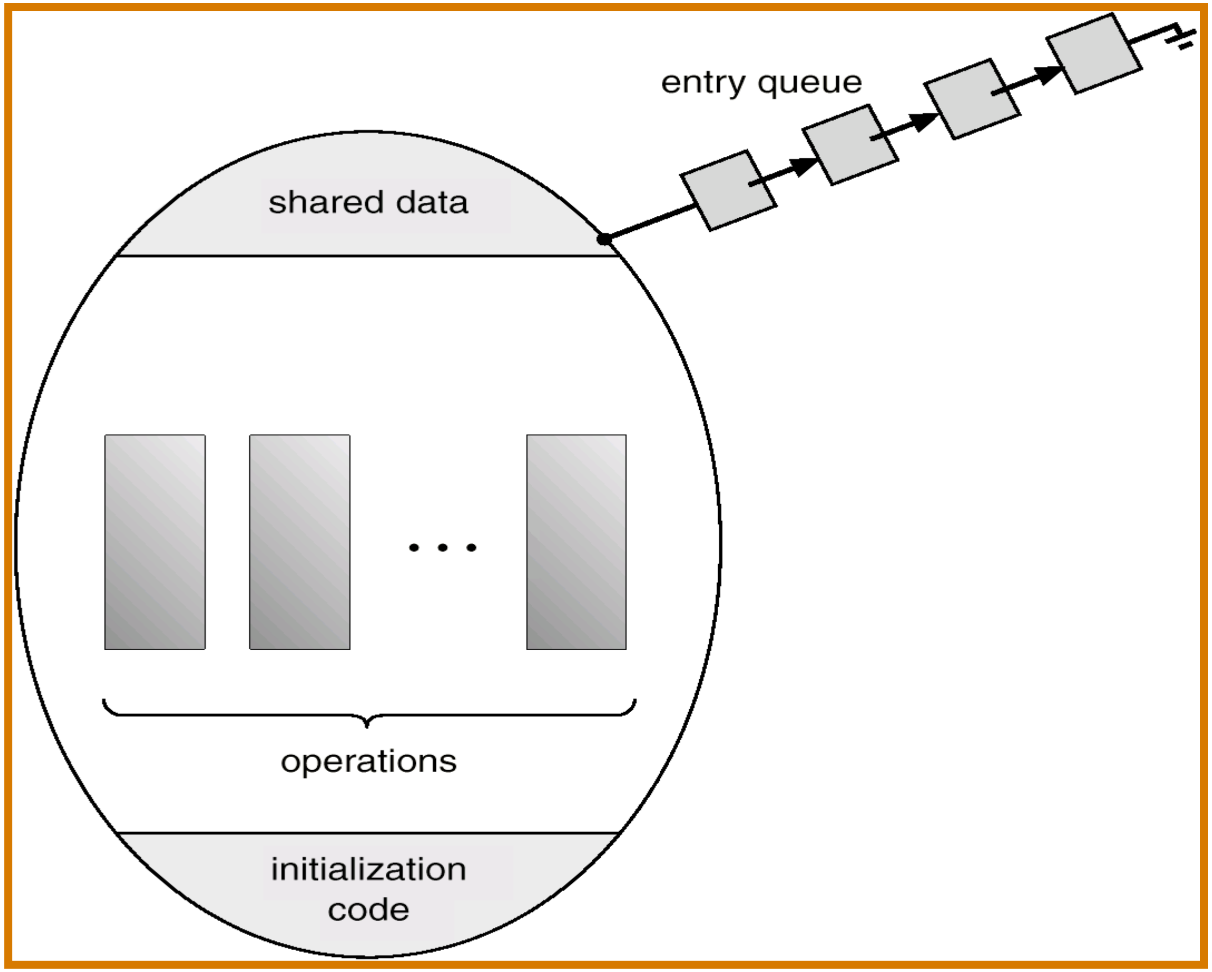
```
monitor mName {
  // shared variables declaration
  procedure P1 (…) {
     …
  }
  procedure Pn (…) {
     …
  }
  init code (…) {
    ….
  }
}
```

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.

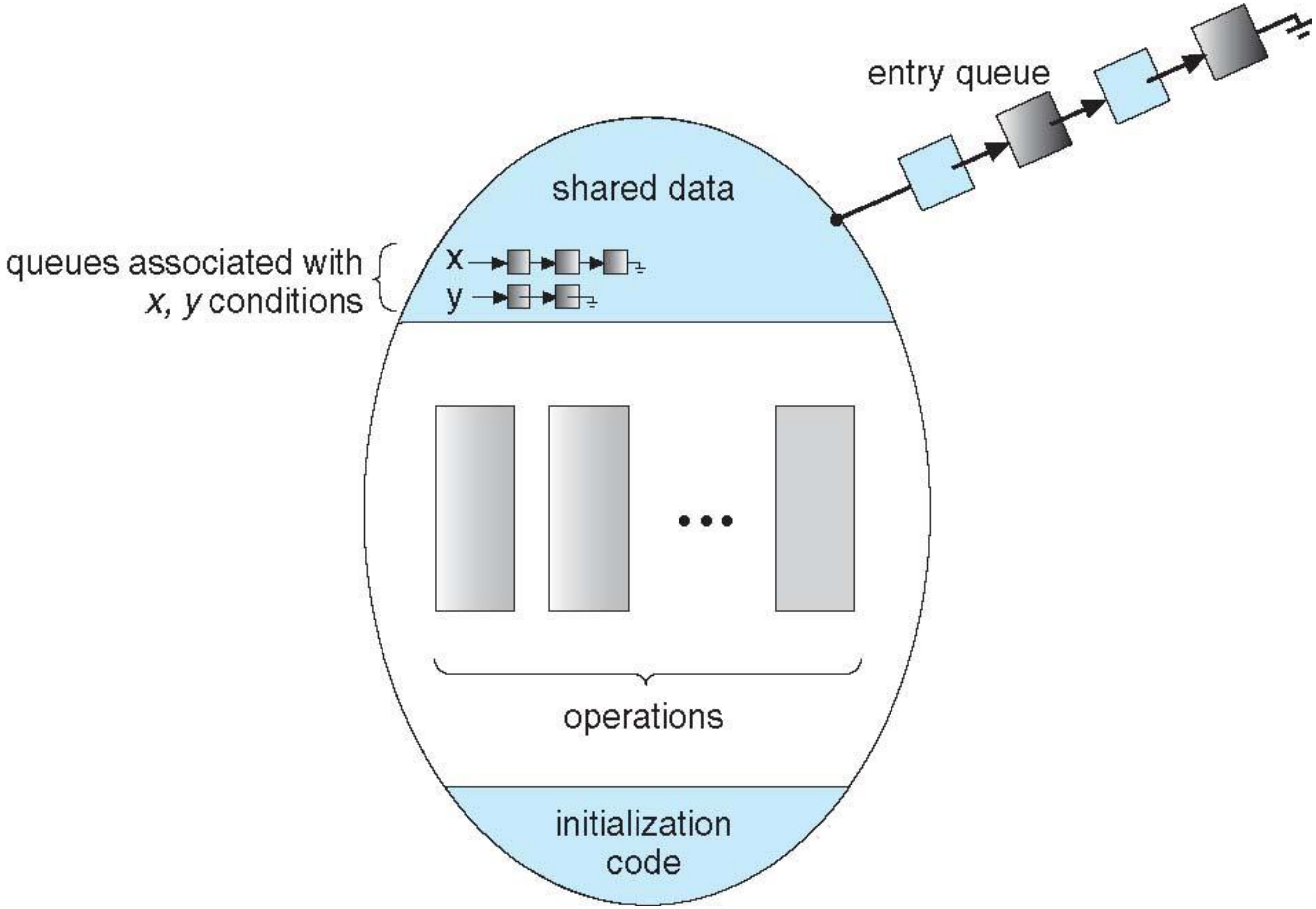
Monitor



Condition Variables

- **`condition x, y;`**
- Two operations are allowed on a condition variable:
  - **`x.wait()`** – a process that invokes the operation is suspended until **`x.signal()`**
  - **`x.signal()`** – resumes one of processes (if any) that invoked **`x.wait()`**
    - If no **`x.wait()`** on the variable, then it has no effect on the variable

Monitor with Condition Variables



Condition Variables Choices

- If process P invokes **`x.signal()`**, and process Q is suspended in **`x.wait()`**, what should happen next?
  - Both Q and P cannot execute in paralel. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide

Deadlock and Starvation

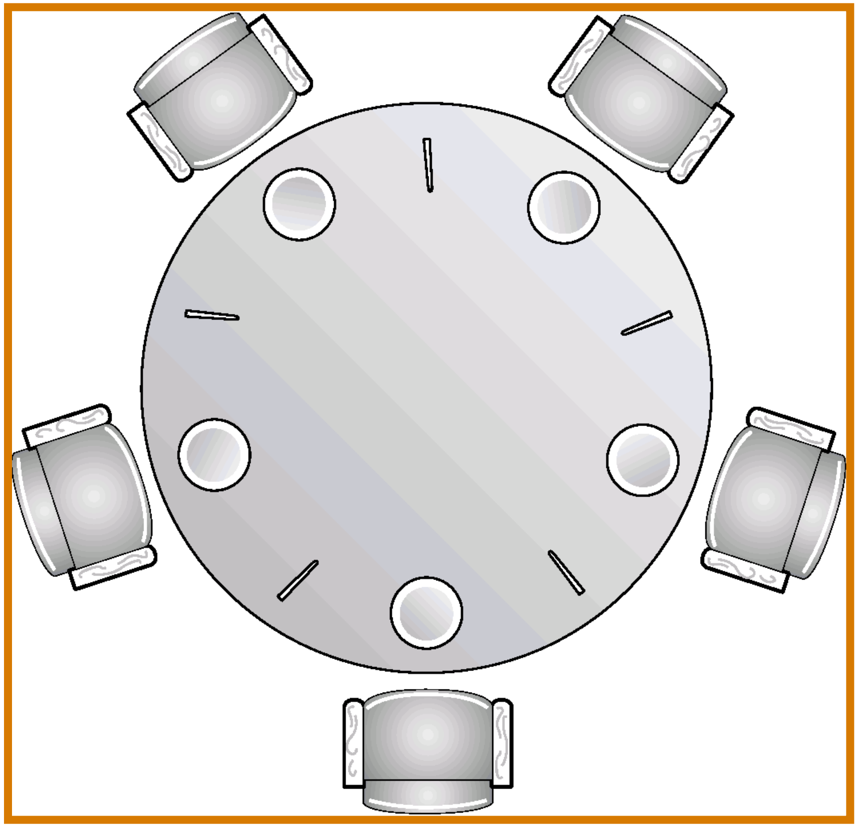
- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

- Let S and Q be two semaphores initialized to 1

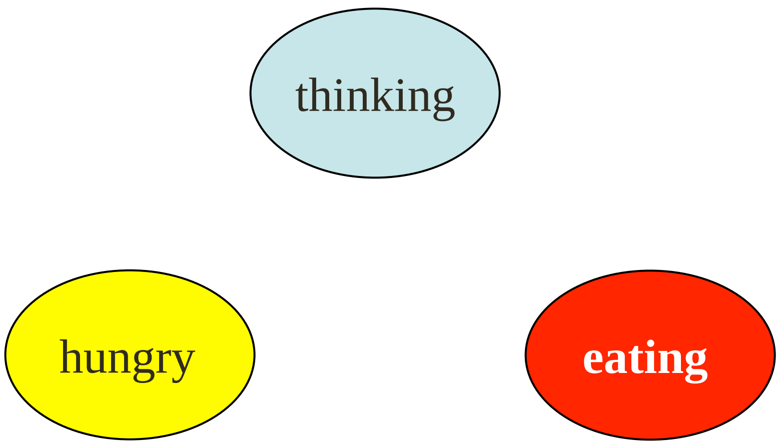
P_0	P_1
acquire(S);	acquire(Q);
acquire(Q);	acquire(S);
.	.
.	.
.	.
release(S);	release(Q);
release(Q);	release(S);

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

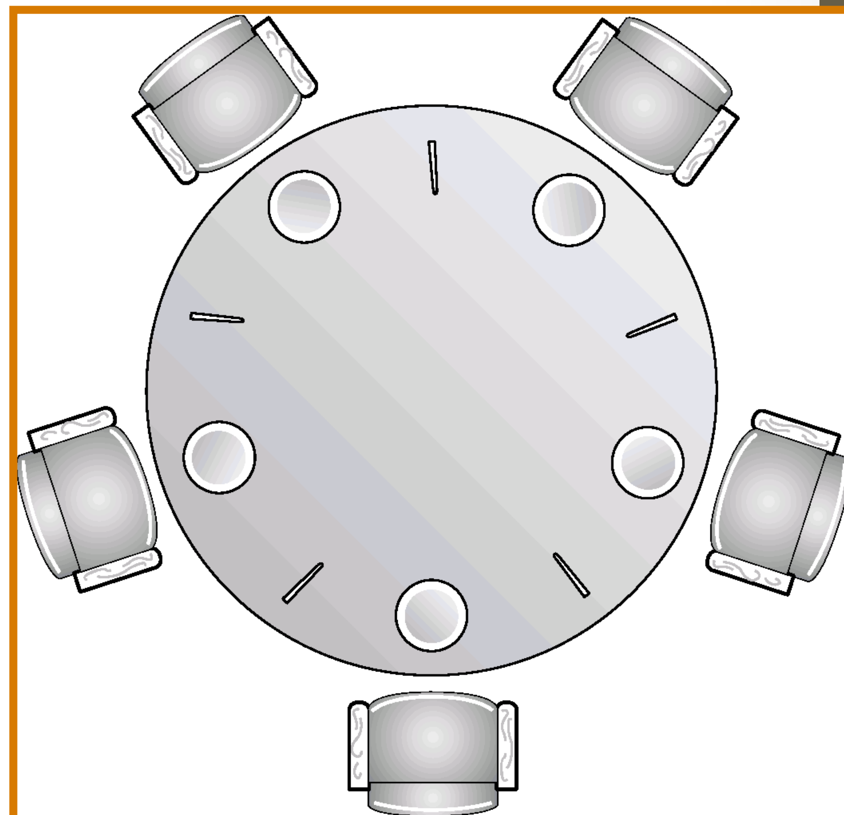
The *Dining-Philosophers* Problem



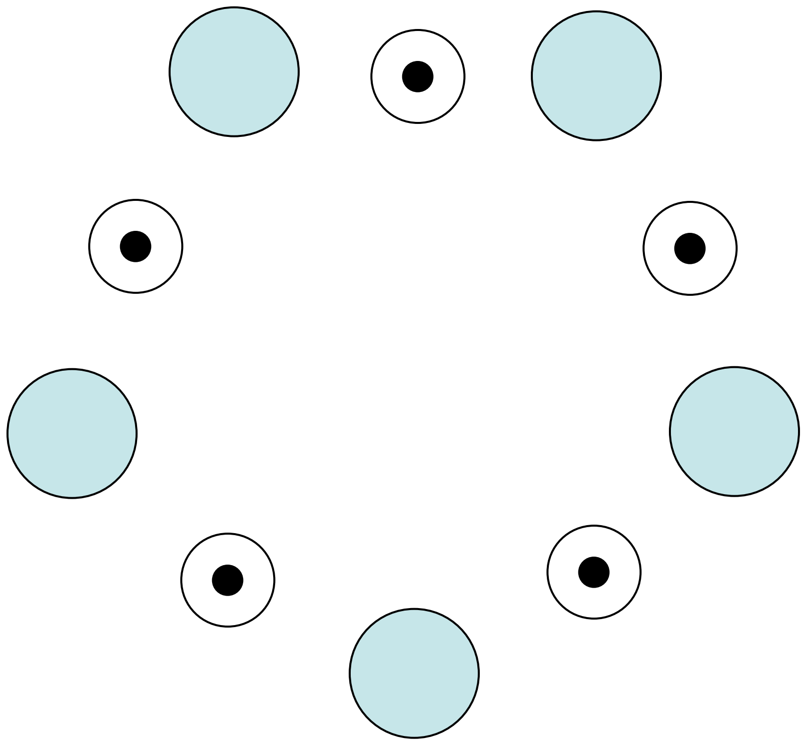
The *Dining-Philosophers* Problem



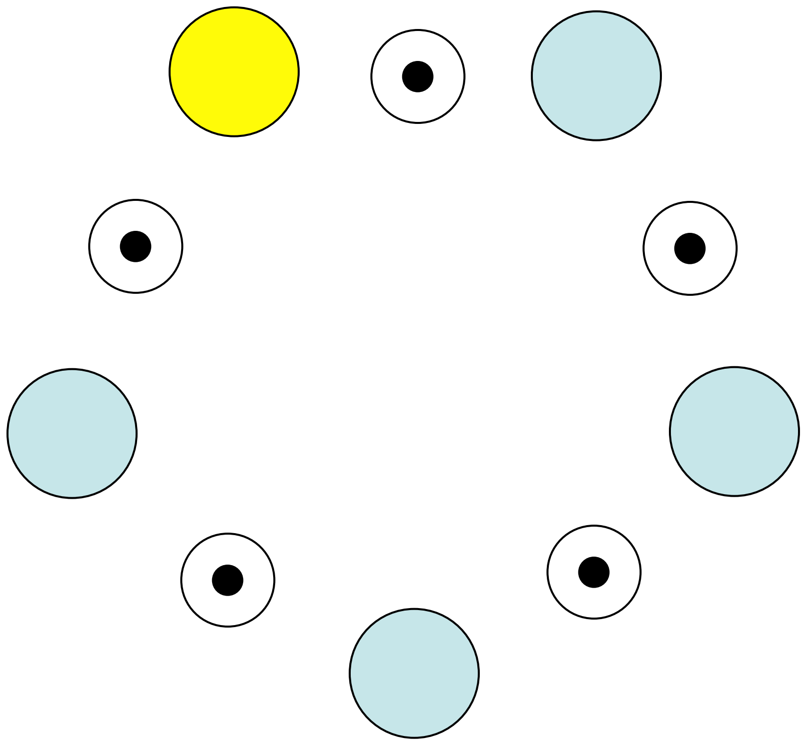
States for a philosopher



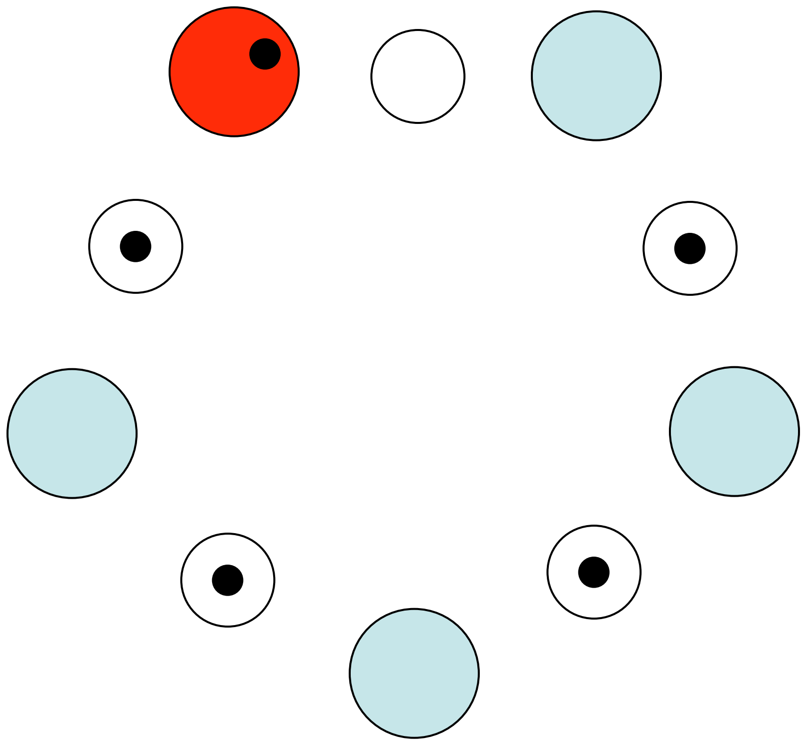
The *Dining-Philosophers* Problem



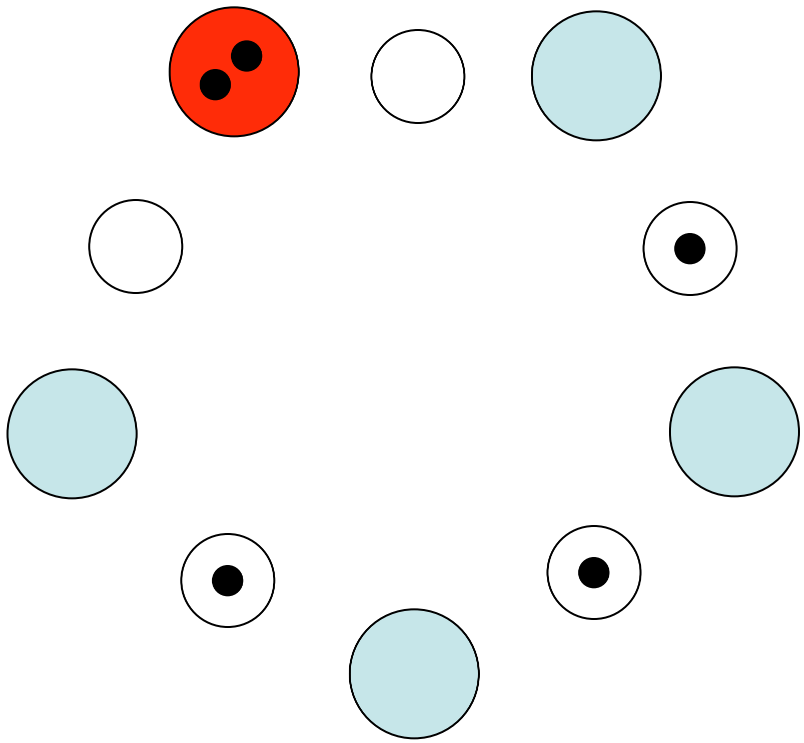
The *Dining-Philosophers* Problem



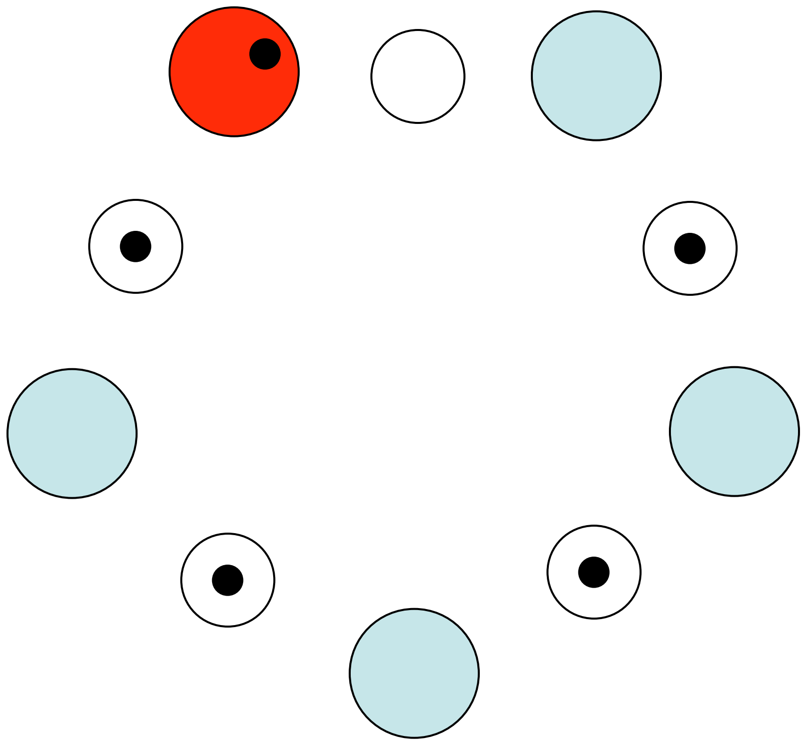
The *Dining-Philosophers* Problem



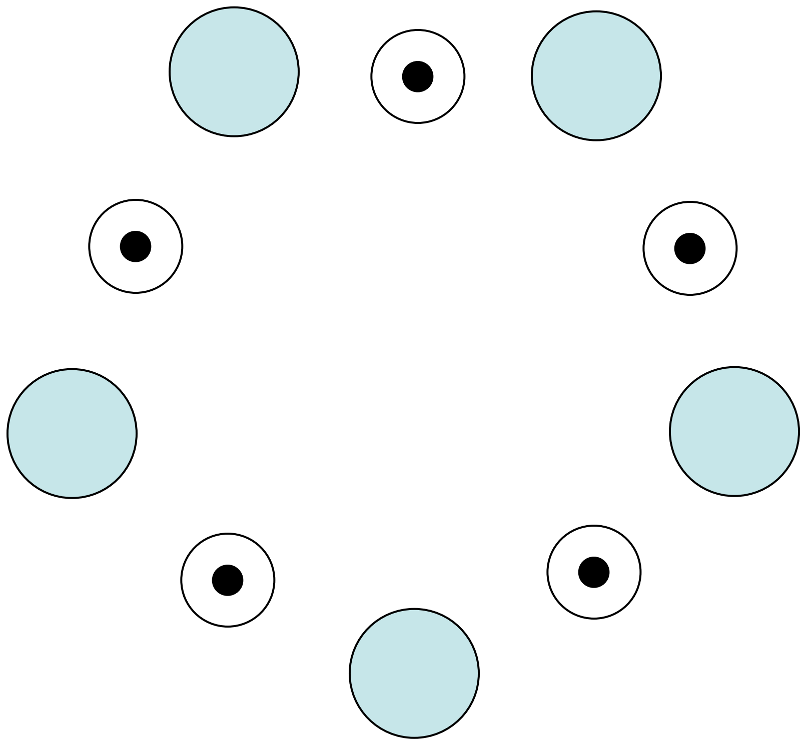
The *Dining-Philosophers* Problem



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?

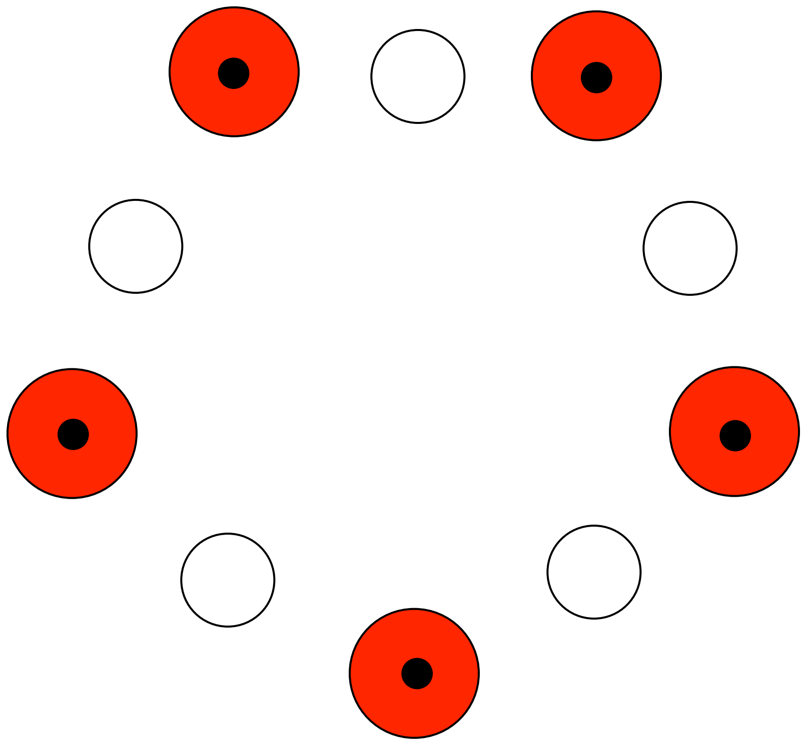
Dining-Philosophers Problem Algorithm

- 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?

The *Dining-Philosophers* Problem



How can we resolve this deadlock?

The *Dining-Philosophers* Problem

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

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

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

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;
    }
}
```