BUCKNELL UNIVERSITY
Computer Science

CSCI 315 Operating Systems Design

Process Synchronization

Notice: The slides for this lecture have been largely based on those accompanying an earlier version of the course text *Operating Systems Concepts with Java*, by Silberschatz, Galvin, and Gagne. Many, if not all, the illustrations contained in this presentation come from this source.

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).
- Note that one can implement a counting semaphore *S* as a binary semaphore.
- Provides **mutual exclusion**:

```
Semaphore S(1); // initialized to 1

acquire(S);
criticalSection();
release(S);
```

Semaphore Implementation

```
acquire(S) {
    value--;
    if (value < 0) {
        add this process to list
        block;
    }
}
```

```
release(S) {
    value++;
    if (value <= 0) {
        remove some process P
          from list
        wakeup(P);
    }
}
```

Semaphore Implementation

- Must guarantee that no two processes can execute acquire() and release() on the same semaphore at the same time.

- The implementation becomes the critical section problem:
  - Could now have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
  - Applications may spend lots of time in critical section

Deadlock and Starvation

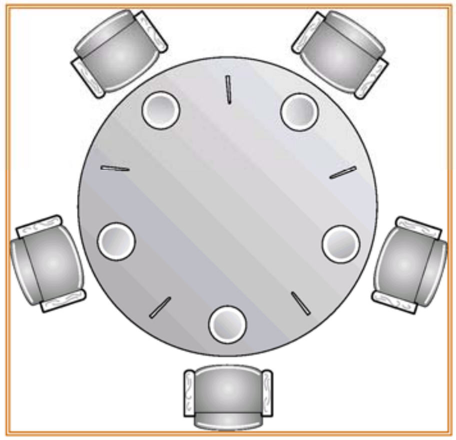
- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

- Let S and Q be two semaphores initialized to 1

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

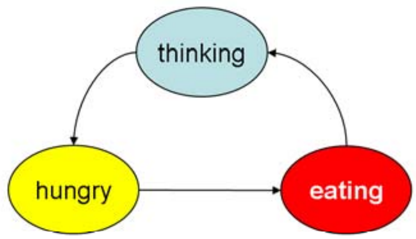
- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

The *Dining-Philosophers* Problem

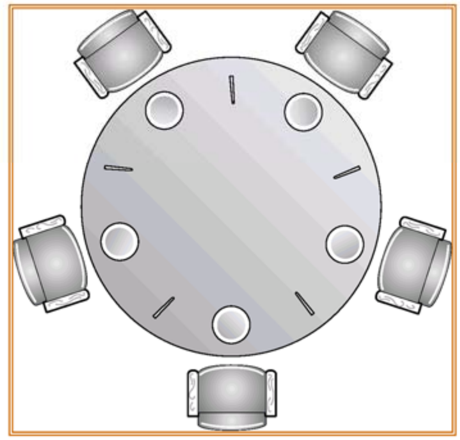


02/19/2010 CSCI 315 Operating Systems Design 6

The *Dining-Philosophers* Problem



State diagram for a philosopher



Shared data:

semaphore chopstick[5];

(Initially all values are 1)

The *Dining-Philosophers* Problem

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

Dining-Philosophers Solution?

<u>**Philosopher *i***</u>

```
do {
   wait(chopstick[i])
   wait(chopstick[(i+1) % 5])
      …
      eat
      …
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      …
      think
      …
   } while (1);
```

Monitor

Definition: High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.

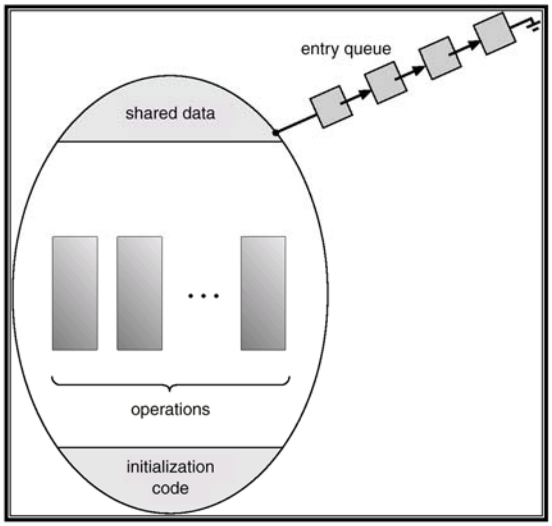
```
monitor monitor-name
{
    shared variables
    procedure body P1 (…) {
            . . .
    }
    procedure body P2 (…) {
            . . .
    }
    procedure body Pn (…) {
            . . .
    }
    {
            initialization code
    }
}
```

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

There cannot be concurrent access to procedures within the monitor (only one 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**.

Schematic View of a Monitor



Monitor

- To allow a process to wait within the monitor, a **condition** variable must be declared, as

  condition x, y;

- Condition variable can only be used with the operations **wait** and **signal**.
  - The operation

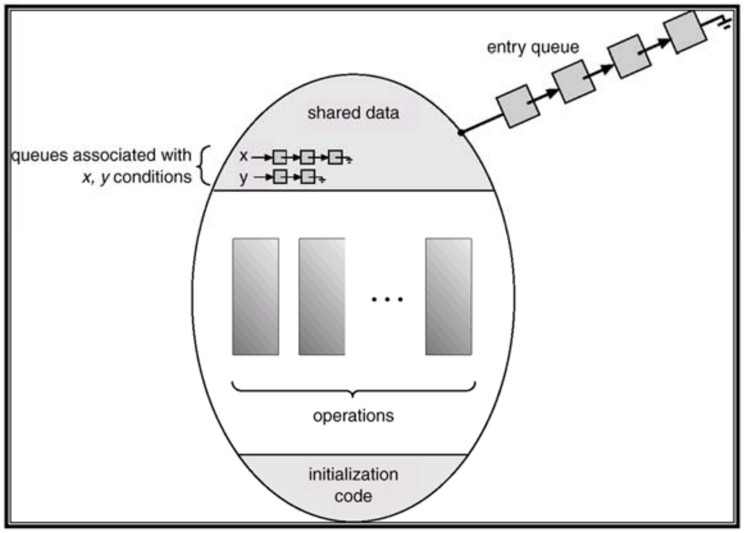
    x.wait();

    means that the process invoking this operation is suspended until another process invokes

    x.signal();

  - The **x.signal** operation resumes exactly one suspended process. If no process is suspended, then the **signal** operation has no effect.

Monitor and Condition Variables



Dining Philosophers with Monitor

```
monitor dp
{
  enum {thinking, hungry, eating} state[5];
  condition self[5];
  void pickup(int i);
  void putdown(int i);
  void test(int i);
  void init() {
       for (int i = 0; i < 5; i++)
              state[i] = thinking;
  }
}
```

Dining Philosophers

```
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);
}
```

```
void test(int i) {
    if ( (state[(i + 4) % 5] != eating) &&
        (state[i] == hungry) &&
        (state[(i + 1) % 5] != eating)) {
              state[i] = eating;
              self[i].signal();
    }
}
```