Operating System

Lecture 11: Process Synchronization Cont.

Manoj Kumar Jain

M.L. Sukhadia University Udaipur

Outline

- Classical Problems of Synchronization
- Monitors

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

- Shared data

  semaphore full, empty, mutex;

  Initially:

  full = 0, empty = n, mutex = 1

Bounded-Buffer Problem Producer Process

do {

...

produce an item in **nextp**

...

wait(empty);

wait(mutex);

...

add **nextp** to buffer

...

signal(mutex);

signal(full);

} while (1);

Bounded-Buffer Problem Consumer Process

```
do {
    wait(full)
    wait(mutex);
        …
    remove an item from buffer to nextc
        …
    signal(mutex);
    signal(empty);
        …
    consume the item in nextc
        …
} while (1);
```

Readers-Writers Problem

- Shared data

  semaphore mutex, wrt;

  Initially

  mutex = 1, wrt = 1, readcount = 0

Readers-Writers Problem Writer Process

wait(wrt);

...

writing is performed

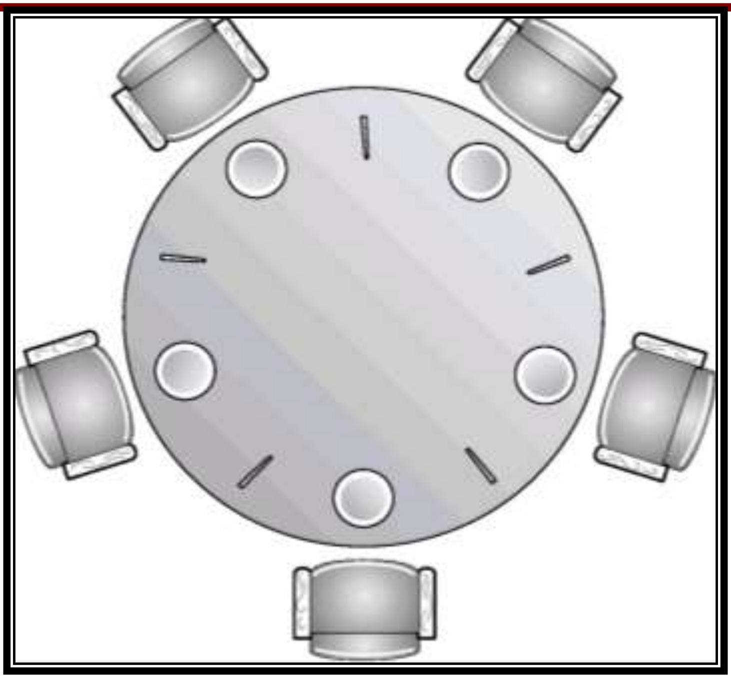
...

signal(wrt);

Readers-Writers Problem Reader Process

```
wait(mutex);
readcount++;
if (readcount == 1)
      wait(rt);
signal(mutex);
       …
   reading is performed
       …
wait(mutex);
readcount--;
if (readcount == 0)
   signal(wrt);
signal(mutex):
```

Dining-Philosophers Problem



- Shared data

semaphore chopstick[5];

Initially all values are 1

Dining-Philosophers Problem

- Philosopher *i*

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

Monitors

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

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

Monitors

- 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

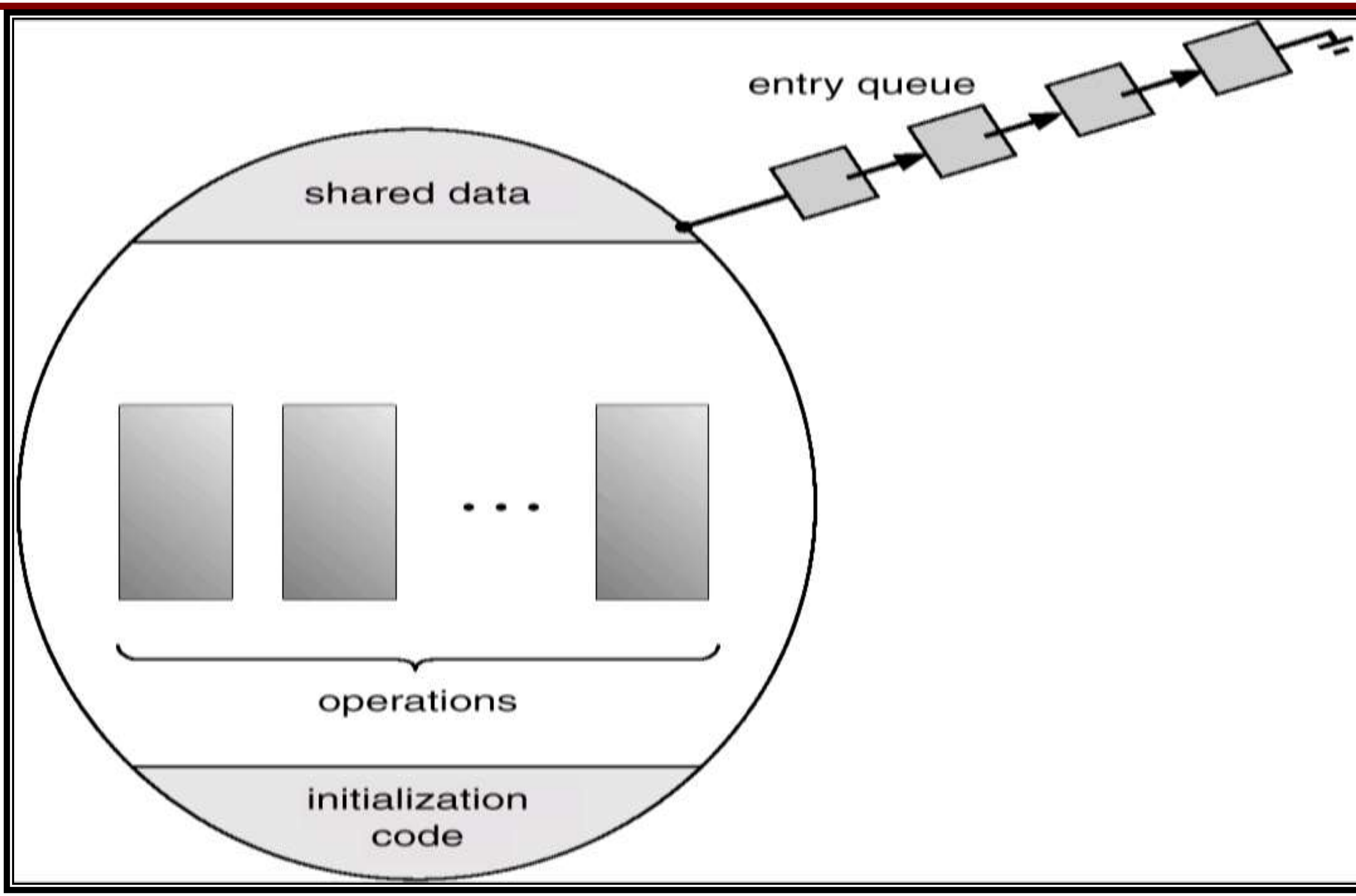
    x.wait();

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

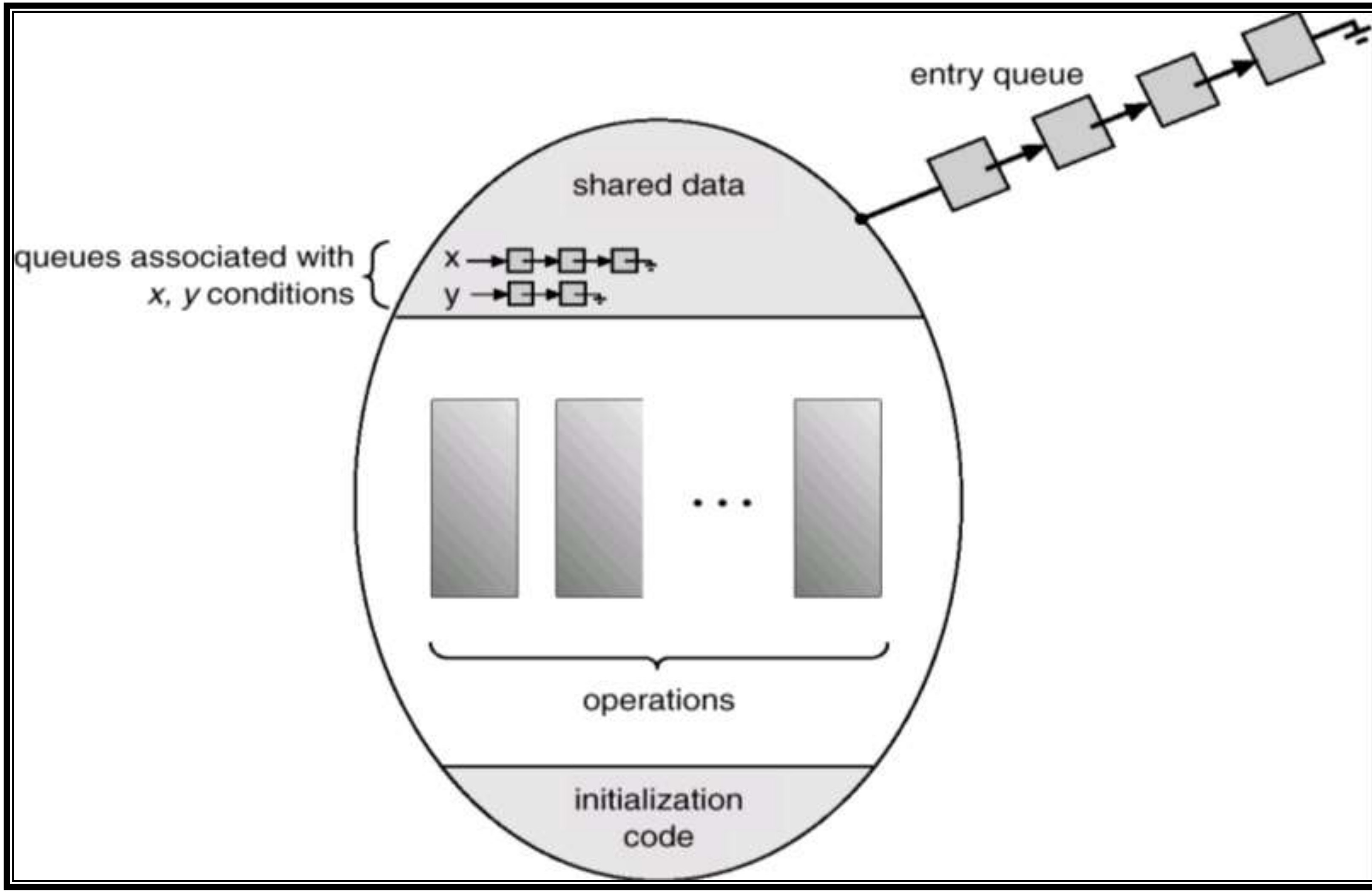
    x.signal();

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

Schematic View of a Monitor



Monitor With Condition Variables



Dining Philosophers Example

```
monitor dp
{
  enum {thinking, hungry, eating} state[5];
  condition self[5];
  void pickup(int i)                    // following
slides
  void putdown(int i)        // following slides
  void test(int i)           // following slides
  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);
}
```

Dining Philosophers

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

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next-count = 0;
```

- Each external procedure ***F*** will be replaced by

```
wait(mutex);
    …
  body of F;
    …
if (next-count > 0)
  signal(next)
else
  signal(mutex);
```

- Mutual exclusion within a monitor is ensured.

Monitor Implementation

- For each condition variable ***x***, we have:

```
semaphore x-sem; // (initially  = 0)
int x-count = 0;
```

- The operation **x.wait** can be implemented as:

```
x-count++;
if (next-count > 0)
    signal(next);
else
    signal(mutex);
wait(x-sem);
x-count--;
```

Monitor Implementation

- The operation **x.signal** can be implemented as:

```
if (x-count > 0) {
   next-count++;
   signal(x-sem);
   wait(next);
   next-count--;
}
```

Monitor Implementation

- *Conditional-wait* construct: **x.wait(c);**
  - **c** – integer expression evaluated when the **wait** operation is executed.
  - value of **c** (a *priority number*) stored with the name of the process that is suspended.
  - when **x.signal** is executed, process with smallest associated priority number is resumed next.
- Check two conditions to establish correctness of system:
  - User processes must always make their calls on the monitor in a correct sequence.
  - Must ensure that an uncooperative process does not ignore the mutual-exclusion gateway provided by the monitor, and try to access the shared resource directly, without using the access protocols.

Thanks