Synchronization

Motivation

- Concurrent processes accessing shared data
  - May result in data inconsistency
- Example reconsider the producer-consumer problem
  - Simplifying assumption: A single processor system
  - Remember the circular-array implementation of buffer
  - Want to use full buffer
    - Use a shared variable **`counter`** which keeps track of number of items in the buffer

Producer

```
while (true) {
   /* produce an item in next produced */
   while (counter == BUFFER SIZE)
      ; /* do nothing */

   buffer[in] = next produced;
   in = (in + 1) % BUFFER SIZE;
   counter++;
}
```

Consumer

```
while (true) {
   while (counter == 0)
      ; /* do nothing */

   next consumed = buffer[out];
   out = (out + 1) % BUFFER SIZE;
   counter--;
   /* consume the item in next consumed */
}
```

Implementation of `counter++` and `counter--`

$register1 =$ **`counter`**
$register1 = register1 + 1$
`counter` $= register1$

`counter++`

$register2 =$ **`counter`**
$register2 = register2 - 1$
`counter` $= register2$

`counter--`

Race conditions

- Assuming the current value of counter is 5
  - What will be the value if both producer and consumer run concurrently?
  - Answer: 4, 5 or 6
  - Why?

Race conditions

$T0$: *producer* execute $register1$ = counter {$register1 = 5$}
$T1$: *producer* execute $register1 = register1 + 1$ {$register1 = 6$}
$T2$: *consumer* execute $register2$ = counter {$register2 = 5$}
$T3$: *consumer* execute $register2 = register2 - 1$ {$register2 = 4$}
$T4$: *producer* execute counter = $register1$ {$counter = 6$}
$T5$: *consumer* execute counter = $register2$ {$counter = 4$}

The Critical Section Problem

- Each process has a so called critical section
  - Changes the value of shared data
- Critical section problem: design a protocol that processes can use to cooperate

Structure of a process

```
do {
    entry section

        critical section

    exit section

        remainder section

} while (true);
```

Constraints on solutions to the Critical Section Problem

- Mutual exclusion
  - Exactly one process may be executing in its critical section
- Progress
  - Only processing requesting entry into critical section may be considered for entry
- Bounded waiting
  - Limit on the number of times entry request may be deferred

Race conditions in the kernel

- Preemptive kernels
  - Possibility of race conditions – need an avoidance protocol
  - Better system performance – a kernel process is not allowed to hold on to the CPU for too long
- Nonpreemptive kernels
  - No possibility of race conditions on single processor systems
  - Poor system performance

Peterson’s Solution

- Restricted to two process that take turns in executing their critical sections

```
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);

      critical section

   flag[i] = false;

      remainder section

} while (true);
```

Synchronization hardware

- Locking
- One possibility is to disable interrupts
  - Only works for single processor systems
  - Inefficient for multicore systems – message has to be passed to all processors
  - Aversely affects system clock, if it’s update is interrupt based
- Modern CPU architectures provide two atomic (uninterruptible) operations
  - **`test_and_set()`**
  - **`compare_and_swap()`**
  - Can use these to solve the critical section problem

test_and_set()

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;
   return rv;
}
```

- target is a pointer (reference) parameter
  - Changes to target are reflected in the actual parameter

test_and_set()

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */
      /* critical section */
   lock = false;
      /* remainder section */
} while (true);
```

compare_and_swap()

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

compare_and_swap()

```
do {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */
      /* critical section */
   lock = 0;
      /* remainder section */
} while (true);
```

Synchronization hardware

- The two algorithms just presented
  - Guarantee mutual exclusion
  - Do not guarantee bounded waiting
- Next algorithm guarantees both

```
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;
    else
        waiting[j] = false;

        /* remainder section */

} while (true);
```

Mutex locks

- **Mutex** (= **mut**ual **ex**clusion) lock
  - Software mechanism available to application programmers
  - **`acquire()`** – process must successful execute this before entering CS
  - **`release()`** – process must execute this on exit to free the lock
  - **`available`** – Boolean indicating whether the lock is available

Mutex locks

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

   available = false;;
}

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

Mutex locks

```
do {
   acquire lock
      critical section
   release lock
      remainder section
} while (true);
```

Mutex locks

- Disadvantage
  - busy waiting – spinlock
    - Wastes CPU cycles – a problem in a multiprogramming environment
- Spinlocks
  - Can be useful for short waiting time
    - When context-switching would consume more time

Semaphores

- Semaphore is an integer + two atomic operations
  - wait()
  - signal()
- Binary semaphore – domain is {0, 1}
  - Same functionality as a mutex lock
- Counting semaphore – unrestricted domain
  - Controls to a resource with finite instances

Semaphores

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

   S--;
}

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

Binary semaphore – use case

- Synchronizing two processes
  - Say P2 must statement S2 only after P1 executes statement S1
  - Initialize semaphore synch to 0

```
P1{
   ...
   S1;
   signal(synch)
   ...
p2{
   ...
   wait(synch);
   S2
```

Semaphore + blocking `wait()`

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

Linked list of processes waiting on the semaphore

Semaphore + blocking **wait()**

```
wait(semaphore *S) {
  S->value--;

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

Semaphore + blocking **`wait()`**

```
signal(semaphore *S) {
   S->value++;

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

Semaphores – blocking **`wait()`**

- Both **`wait()`** and **`signal()`** must be executed atomically
  - Critical sections!
- Atomicity can be guaranteed by
  - Disabling interrupts – sensible only on single processor systems
  - **Use `test_and_set()`** or **`compare_and_swap()`**
- Blocking wait does not completely eliminate busy waiting

Deadlocks

- **Defn**: A set of process (two or more processes in the set) are deadlocked if each process in the set is waiting for another process in the set to execute signal()
  - Since the process that’s supposed to execute signal is blocked, all processes in the set will wait indefinitely.

Deadlock example

P_0

```
wait(S);
wait(Q);
.
.
.
signal(S);
signal(Q);
```

P_1

```
wait(Q);
wait(S);
.
.
.
signal(Q);
signal(S);
```

Indefinite blocking (starvation)

- A process may be indefinitely blocked is the queue on a semaphore in processed in, say, LIFO

Priority Inversion

- Occurs on systems implementing more than two levels of priority
- Example: Assume process L, M and H have low, medium and high priority resp.
  - L is currently executing and using resource R
  - H becomes runnable and also requests R – it blocks since R is locked, so L continues
  - M becomes runnable and preempts L
  - M has been dispatched while H (with a higher priority) has to wait for a resource being held by L

Priority Inversion - prevention

- Restrict priority levels to 2
  - Not practical on most systems
- Priority inheritance
  - Lower priority process holding a resource required by higher priority process temporarily adopts the higher priority

Classical synchronization problems

- Bounded buffer
- Readers-writers
- Dining philosophers

Bounded buffer problem

- Producer-consumer problem using a bounded buffer
- Shared data

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

Producer

```
do {
   . . .
   /* produce an item in next produced */
   . . .
   wait(empty);
   wait(mutex);
   . . .
   /* add next produced to the buffer */
   . . .
   signal(mutex);
   signal(full);
} while (true);
```

Consumer

```
do {
   wait(full);
   wait(mutex);
   . . .
   /* remove an item from buffer to next consumed */
   . . .
   signal(mutex);
   signal(empty);
   . . .
   /* consume the item in next consumed */
   . . .
} while (true);
```

Readers-writers problem

- First readers-writers problem
  - One writer at a time – no readers permitted
  - Multiple readers at a time
  - Readers must not wait for writer to start writing
- Second readers-writers
  - Writer must write as soon as possible – no new reader admitted as soon as writer is ready to write
- Applications:
  - Can distinguish between readers and writers
  - More readers than writers

First readers-writers problem – shared data

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read count = 0;
```

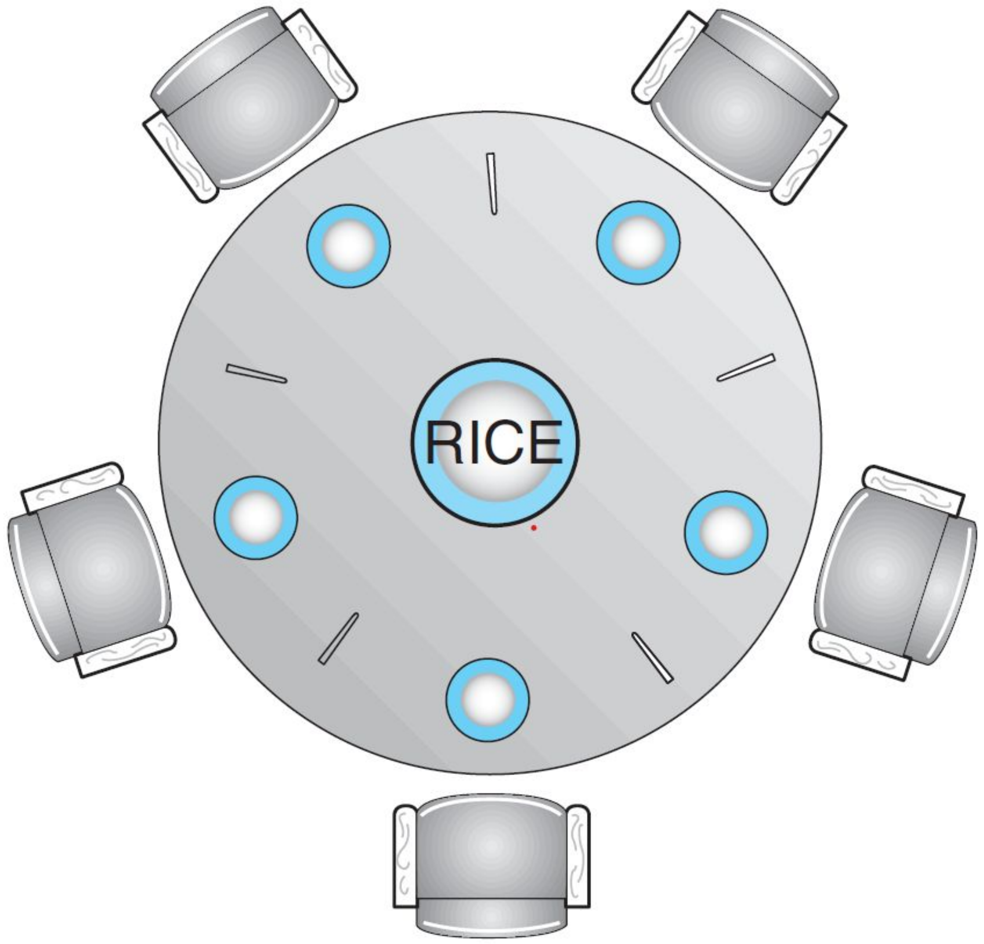
First readers-writers problem - writer

```
do {
   wait(rw_mutex);
   . . .
   /* writing is performed */
   . . .
   signal(rw_mutex);
} while (true);
```

First readers-writers problem - reader

```
do {
   wait(mutex);
   read count++;
   if (read count == 1)
      wait(rw_mutex);
   signal(mutex);
      . . .
      /* reading is performed */
      . . .
   wait(mutex);
   read count--;
   if (read count == 0)
      signal(rw_mutex);
   signal(mutex);
} while (true);
```

Dining philosophers problem



Dining philosophers problem

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

How to deal with deadlocks

- Allows only 4 philosophers to attempt to pick up a chopstick
- Allow a philosopher to attempt to pick up chopsticks only if both are free – critical section
- Asymmetric solution

Weakness of semaphores

- Non-adherence to the **`wait()-signal()`** sequence by processes
    - Deliberate or accidental

```
signal(mutex);
   ...
   critical section
   ...
wait(mutex);
```

- Broken mutual exclusion

Weakness of semaphores

- Non-adherence to the **`wait()-signal()`** sequence by processes
  - Deliberate or accidental

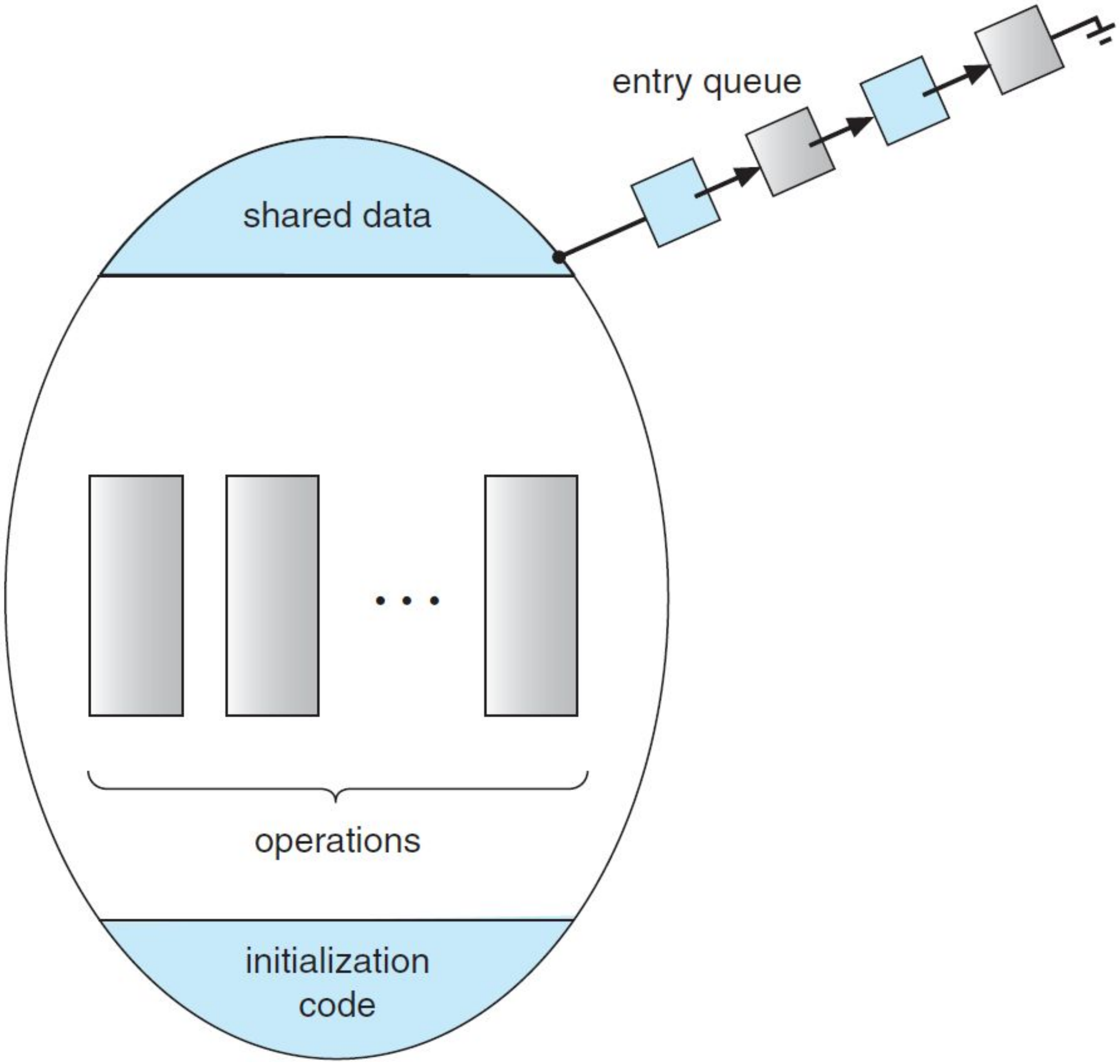
```
wait(mutex);
   ...
   critical section
   ...
wait(mutex);
```

- Deadlock

Monitors

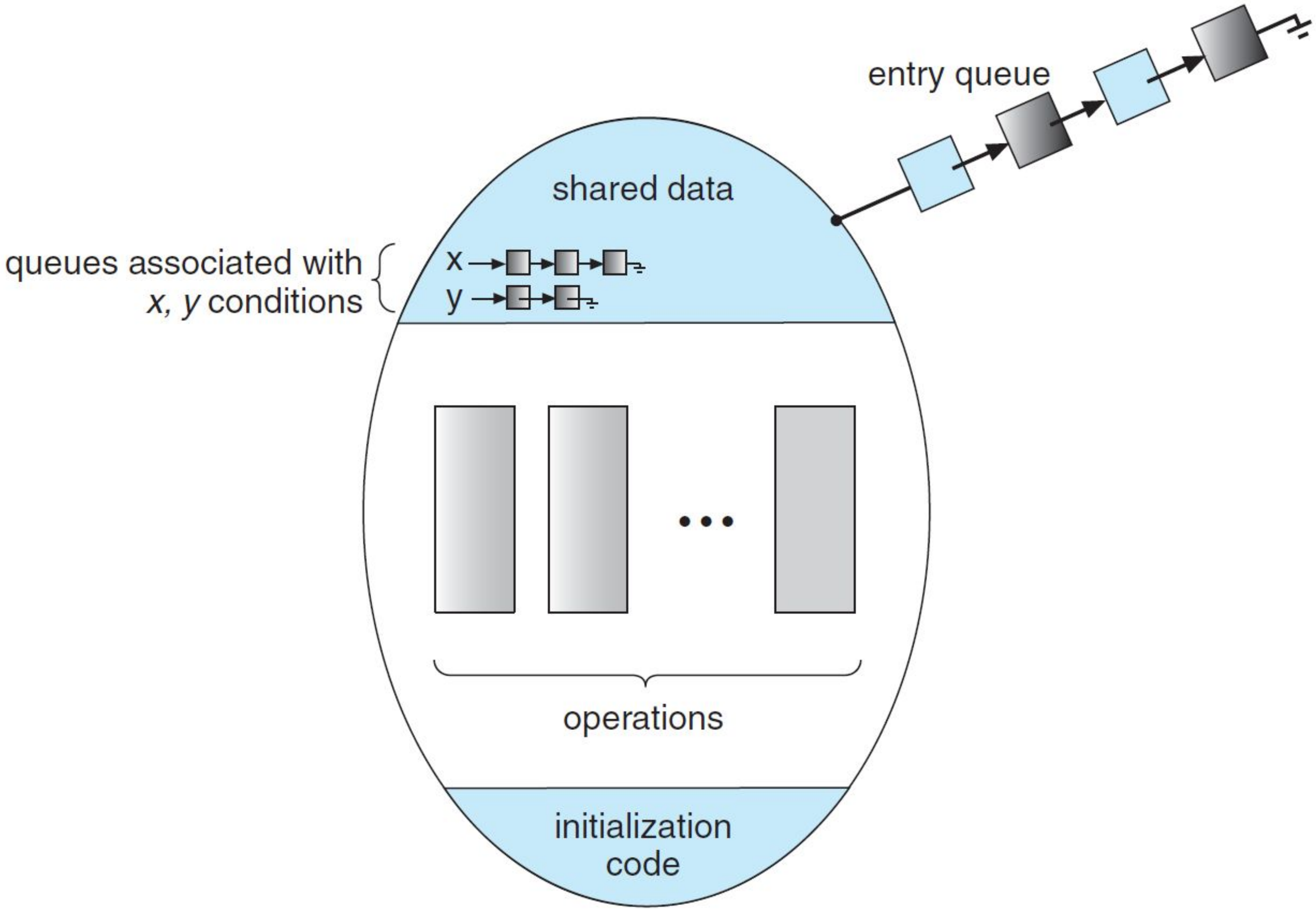
- Programming language support for synchronization
- Solves the semaphore problems
- Available in languages like Java, C#
- Monitor ADT

```
monitor monitor_name{
   /* shared variable declarations */

   function P1 ( . . . ) {
      . . .
   }

   function P2 ( . . . ) {
      . . .
   }
   .
   .
   .
   function Pn ( . . . ) {
      . . .
   }

   initialization_code ( . . . ) {
      . . .
   }
}
```


entry queue
shared data
operations
initialization
code

Monitors

- **condition** type
  - supports two operations
    - **`x.wait()`**
    - **`x.signal()`**
- What happens when process P signals process Q?
  - Either **signal and wait** – P must wait for Q to exit monitor or wait
  - Or **signal and continue** – Q must still wait for P to exit monitor or wait


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

Dining philosophers solution using monitors

```
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((i + 4) % 5);
      test((i + 1) % 5);
   }
   ...
}
```

Dining philosophers solution using monitors

```
monitor DiningPhilosophers{
    ...
    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;
    }
}
```

Dining philosophers solution using monitors

- No two neighbors eat at the same time
- No deadlocks
- Starvation may still occur