Monitors

Semaphores : Disadvantages

- Semaphores Are Not Always Convenient
- Example (adopted from the slides by Kai Li, Computer Science Department, Princeton University)
  - A shared queue has Enqueue and Dequeue:

```
Enqueue(q, item)
{
  Acquire(mutex);
  put item into q;
  Release(mutex);
}
```

```
Dequeue(q)
{
  Acquire(mutex);
  take an item from q;
  Release(mutex);
  return item;
}
```

Semaphores : Disadvantages

- **Problem 1 :** It is a consumer and producer problem, **Dequeue(q)** should block until q is not empty. Results in a lot of waits and signals
- **Problem 2 :** What happens when a programming error changes the order of the wait and signal, or use duplicates wait or signal?

Solution - Monitors

- Higher level solution than Semaphores
- May use semaphore as a low level implementation
- Main theme : hide mutual exclusion!
- Main theme : provide concurrency support in compiler

Monitors

- Consists of -
  - Shared Private Data– cannot be accessed from outside but shared among threads
  - Procedures that operate on the data – Gateway to the resource – can only act on data local to the monitor
  - Synchronization primitives – among threads that access the procedures

Structure of a Monitor

```
Monitor monitor_name
{
      // shared variable declarations

      procedure P1(. . . .) {
          . . . .
      }

      procedure P2(. . . .) {
          . . . .
      }
      :
      procedure PN(. . . .) {
          . . . .
      }

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

For example:

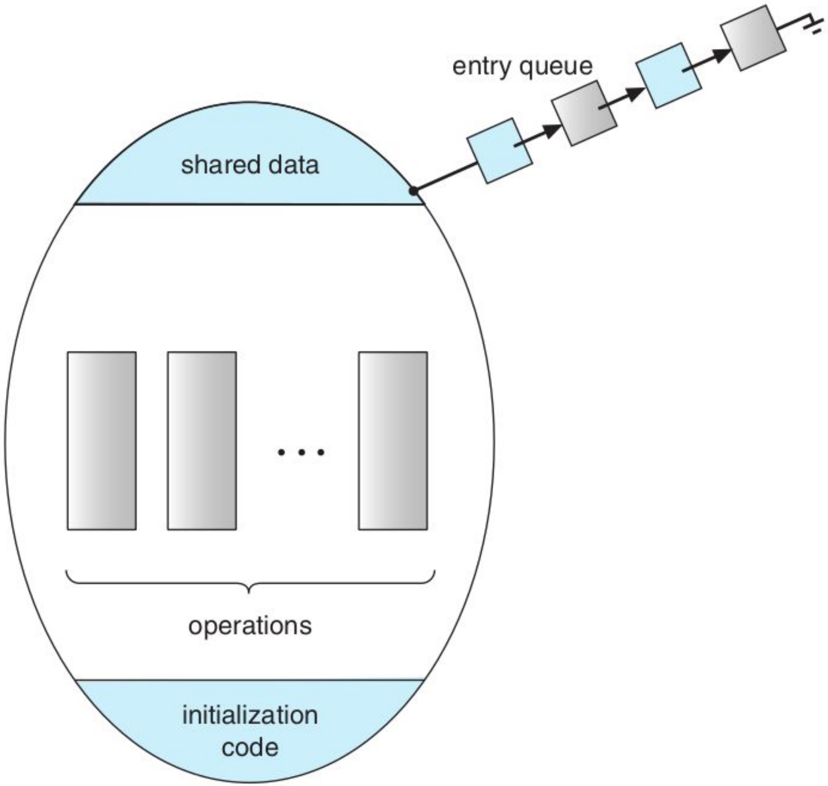
```
Monitor stack
{
      int top;
      void push(any_t *) {
          . . . .
      }

      any_t * pop() {
          . . . .
      }

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

***(adopted from CS 4410 of Cornell University)*

Schematic View of a Monitor



Conditions

- How to develop more complicated synchronization solution?
- Solution : provide wait and signal capability to monitors
- Condition variables can be defined inside monitors
- Wait and signal can be called on these variables
- Condition variables can be considered as a queue inside monitors

Monitors and Conditions Example

```
procedure Producer
begin
  while true do
  begin
    produce an item
    ProdCons.Enter();
  end;
end;

procedure Consumer
begin
  while true do
  begin
    ProdCons.Remove();
    consume an item;
  end;
end;
```

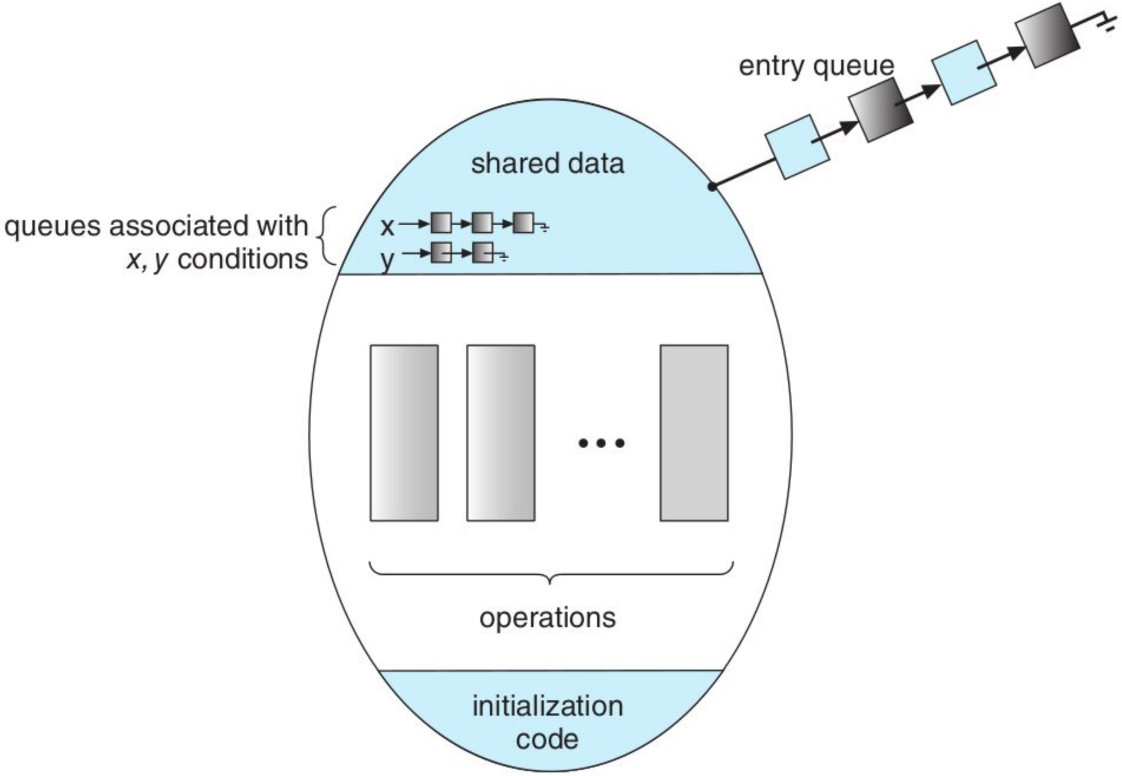
```
monitor ProdCons
  condition full, empty;

  procedure Enter;
  begin
    if (buffer is full)
      wait(full);
    put item into buffer;
    if (only one item)
      signal(empty);
  end;

  procedure Remove;
  begin
    if (buffer is empty)
      wait(empty);
    remove an item;
    if (buffer was full)
      signal(full);
  end;
```

(adopted from the slides by Kai Li, Computer Science Department, Princeton University)

Schematic view of Monitor with Condition Variables



Condition Variables - Wait and Signal Schemes

- Consider P and Q processes using a condition variable x
- Q calls wait(x) and then P calls signal(x)
- Two scenarios can happen -
  - Signal and wait. P either waits until Q leaves the monitor or waits for another condition
  - Signal and continue. Q either waits until P leaves the monitor or waits for another condition
- P was already executing in the monitor, the signal and continue method seems more reasonable
- But, if we allow thread P to continue, then by the time Q is resumed, the logical condition for which Q was waiting may no longer hold
- Solution is to compromise - when thread P executes the signal operation, it immediately leaves the monitor. Hence, Q is immediately resumed

Resuming Processes within a Monitor

- FCFS order
- More complicated ordering - conditional wait
- Conditional wait - x.wait(c), where c is the priority number
- When x.signal() is executed, the process with the smallest priority number is resumed

Resuming Processes within a Monitor

```
monitor ResourceAllocator
{
   boolean busy;
   condition x;

   void acquire(int time) {
      if (busy)
         x.wait(time);
      busy = true;
   }

   void release() {
      busy = false;
      x.signal();
   }

   initialization_code() {
      busy = false;
   }
}
```

```
R.acquire(t);
     ...
   access the resource;
     ...
R.release();
```