

# Monitors

# Monitors

- One of the fundamental high-level synchronization.

## Monitor Usage

- A *monitor type* is an ADT that includes a set of programmer defined operations that are provided with mutual exclusion within the monitor.
- A monitor is a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package.
- Processes may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor.

# Monitors

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


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

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

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

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

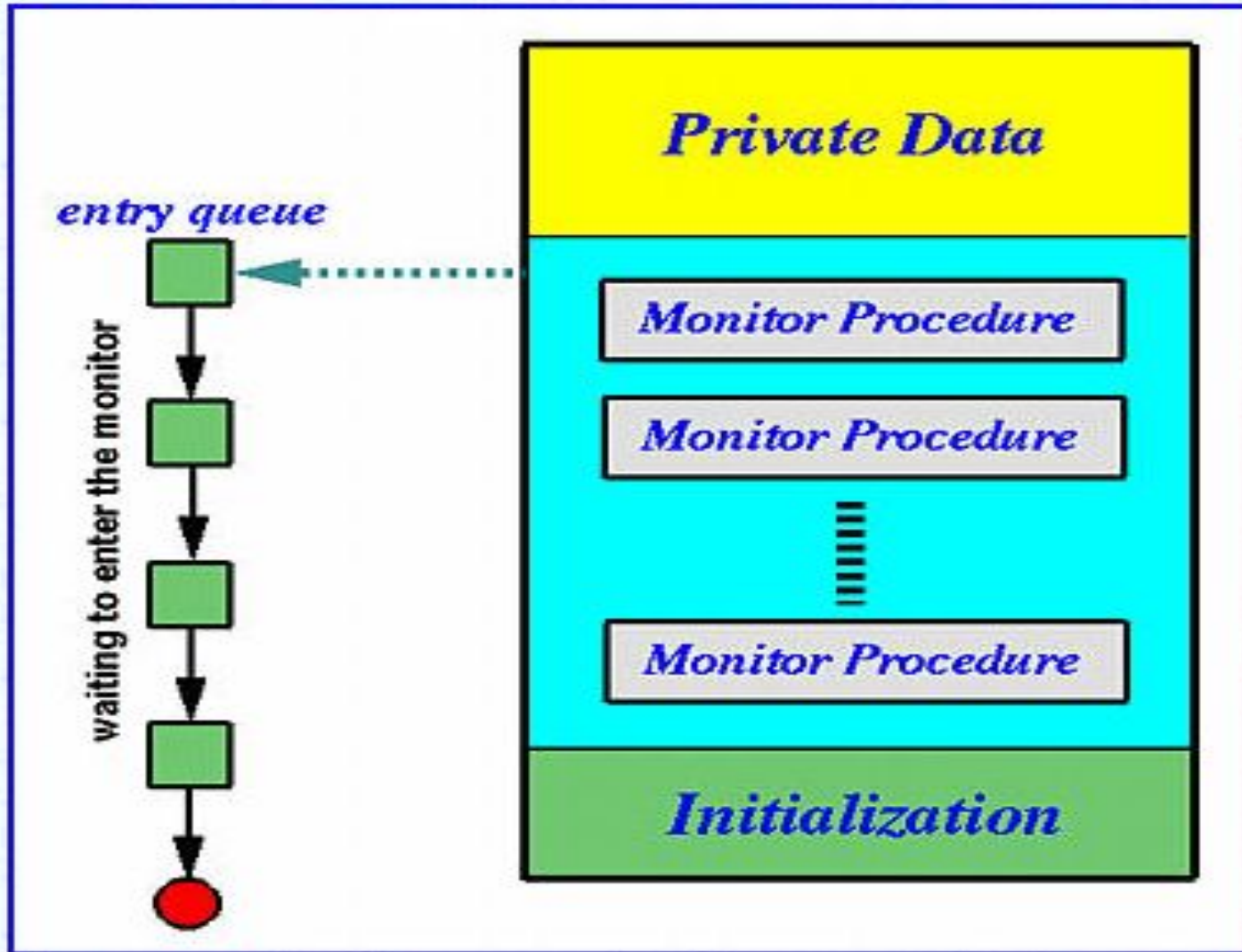
Syntax of a monitor.

- 
- Monitors can be used for achieving mutual exclusion: only one process can be active in a monitor at any instant.
  - Monitors are a programming language construct, so the compiler knows they are special and can handle calls to monitor procedures differently from other procedure calls.
  - When a process calls a monitor procedure, the first few instructions of the procedure will check to see if any other process is currently active within the monitor. If so, the calling process will be suspended until the other process has left the monitor. If no other process is using the monitor, the calling process may enter.

# Monitors

- A monitor has *four* components : initialization, private data, monitor procedures, and monitor entry queue.
- The *initialization* component contains the code that is used exactly once when the monitor is created.
- The *private data* section contains all private data, including private procedures, that can only be used *within* the monitor. These private items are not visible from outside of the monitor.
- The *monitor procedures* are procedures that can be called from outside of the monitor.
- The *monitor entry queue* contains all processes that called monitor procedures but have not been granted permissions.

# Monitors





- **Mutual Exclusion of a Monitor**

- Monitors are used in a multithreaded or multiprocess environment in which multiple threads/processes may call the monitor procedures at the same time asking for service.
- Thus, a monitor guarantees that *at any moment at most one process can be executing in a monitor* the calling process.
- If two processes are in the monitor (*i.e.*, they are executing two, possibly the same, monitor procedures), some private data may be modified by both processes at the same time causing race conditions to occur.

- Therefore, to guarantee the integrity of the private data, a monitor enforces mutual exclusion *implicitly*.
- If a process calls a monitor procedure, this process will be blocked if there is another process executing in the monitor.
- Those process that were not granted the entering permission will be queued to a monitor *entry queue* outside of the monitor.
- When the monitor becomes empty (*i.e.*, no process is executing in it), one of the process in the entry queue will be released and granted the permission to execute the called monitor procedure.
- In summary, monitors ensure mutual exclusion automatically so that there is no more than one process can be executing in a monitor at any time. This is a very usable and handy capability.

- The monitor construct ensures that only one process at a time is active within the monitor.
- Monitor construct, is not sufficiently powerful for modeling some synchronization schemes. To avoid this problem condition construct is used
- condition x, y;
- The only operations that can be invoked on a condition variable are 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.

- Now suppose that, when the `x.signal()` operation is invoked by a process  $P$ , *there exists a suspended process  $Q$  associated with condition  $x$ .*
- *If the suspended process  $Q$  is allowed to resume its execution, the signaling process  $P$  must wait.*
- Two possibilities exist:
  - 1. Signal and wait.**  *$P$  either waits until  $Q$  leaves the monitor or waits for another condition.*
  - 2. Signal and continue.**  *$Q$  either waits until  $P$  leaves the monitor or waits for another condition.*

# Dining-Philosophers Solution Using Monitors

```
enum {THINKING, HUNGRY, EATING} state[5];
```

- Philosopher  $i$  can set the variable  $state[i] = EATING$  only if her two neighbors are not eating:  $(state[(i+4) \% 5] \neq EATING)$  and  $(state[(i+1) \% 5] \neq EATING)$ .
- We also need to declare  
    `condition self[5];`
- This allows philosopher  $i$  to delay herself when she is hungry but is unable to obtain the chopsticks she needs

```

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);
    }

    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;
    }
}

```

A monitor solution to the dining-philosopher problem.

- Each philosopher, before starting to eat, must invoke the operation `pickup()`.
- This act may result in the suspension of the philosopher process.
- After the successful completion of the operation, the philosopher may eat.
- Following this, the philosopher invokes the `putdown()` operation.
- Thus, philosopher *i* *must invoke the operations* `pickup()` and `putdown()` in the following sequence:

`DiningPhilosophers.pickup(i);`

...

`eat`

...

`DiningPhilosophers.putdown(i);`