Operating Systems
CS F372

Synchronization

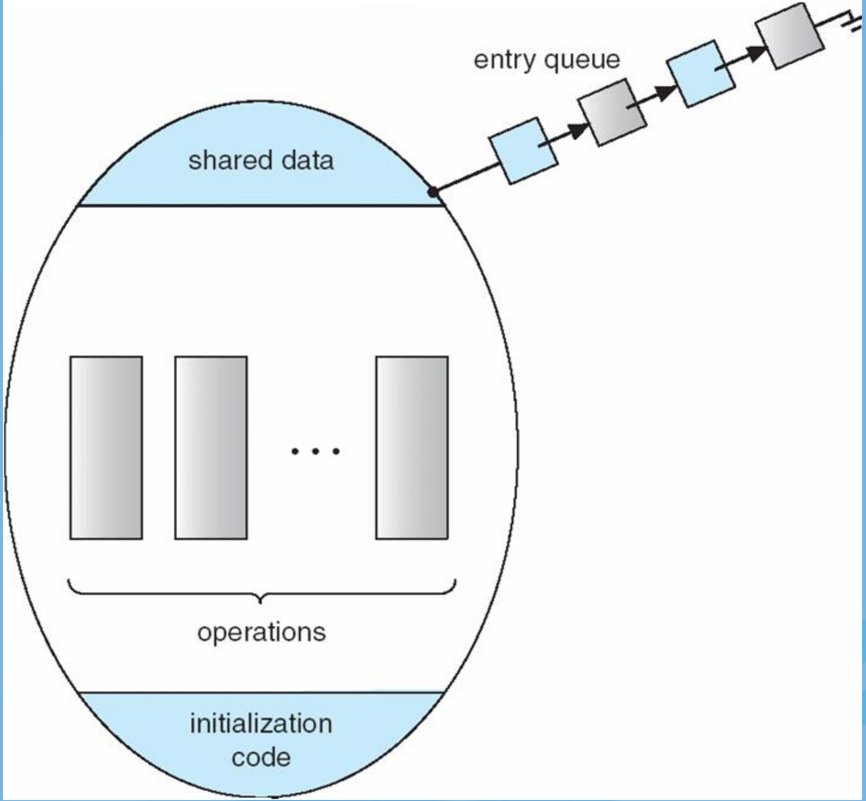
BIJU K RAVEENDRAN

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time
- Process enters monitor by invoking one of its procedures
- Local data variables are accessible only by the monitor

```
monitor monitor-name
{  // shared variable declarations
   procedure P1 (…) { …. }
          …
   procedure Pn (…) {……}
    Initialization code ( ….) { … }
          …
   }
}
```

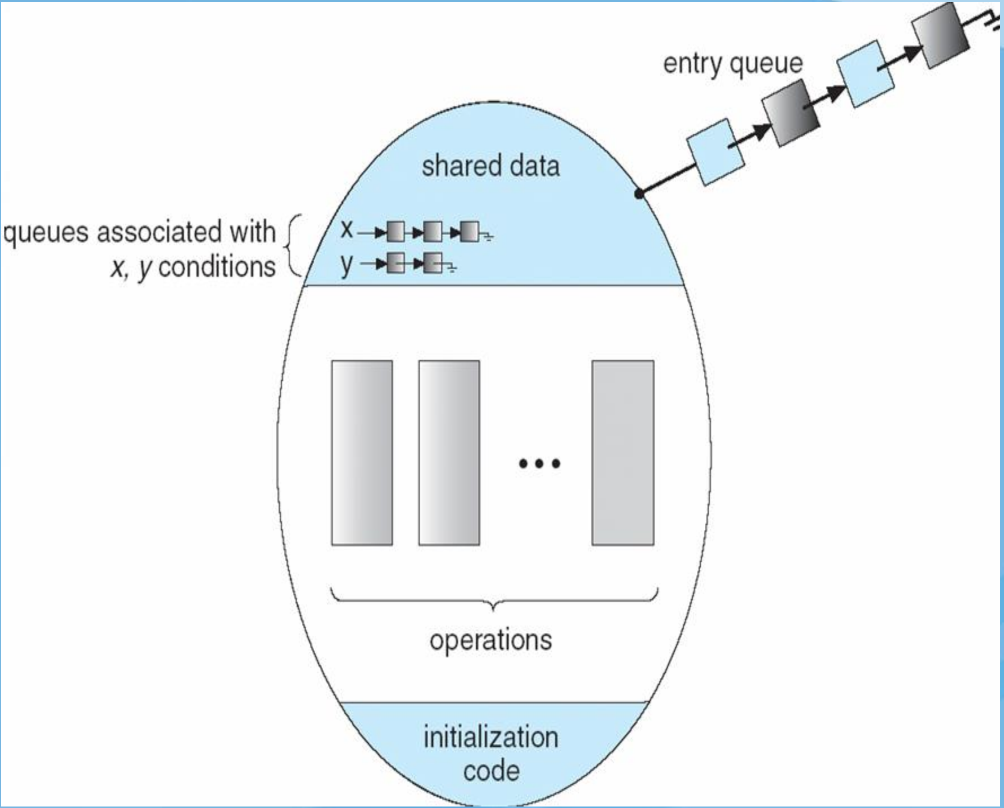
Schematic View of a Monitor

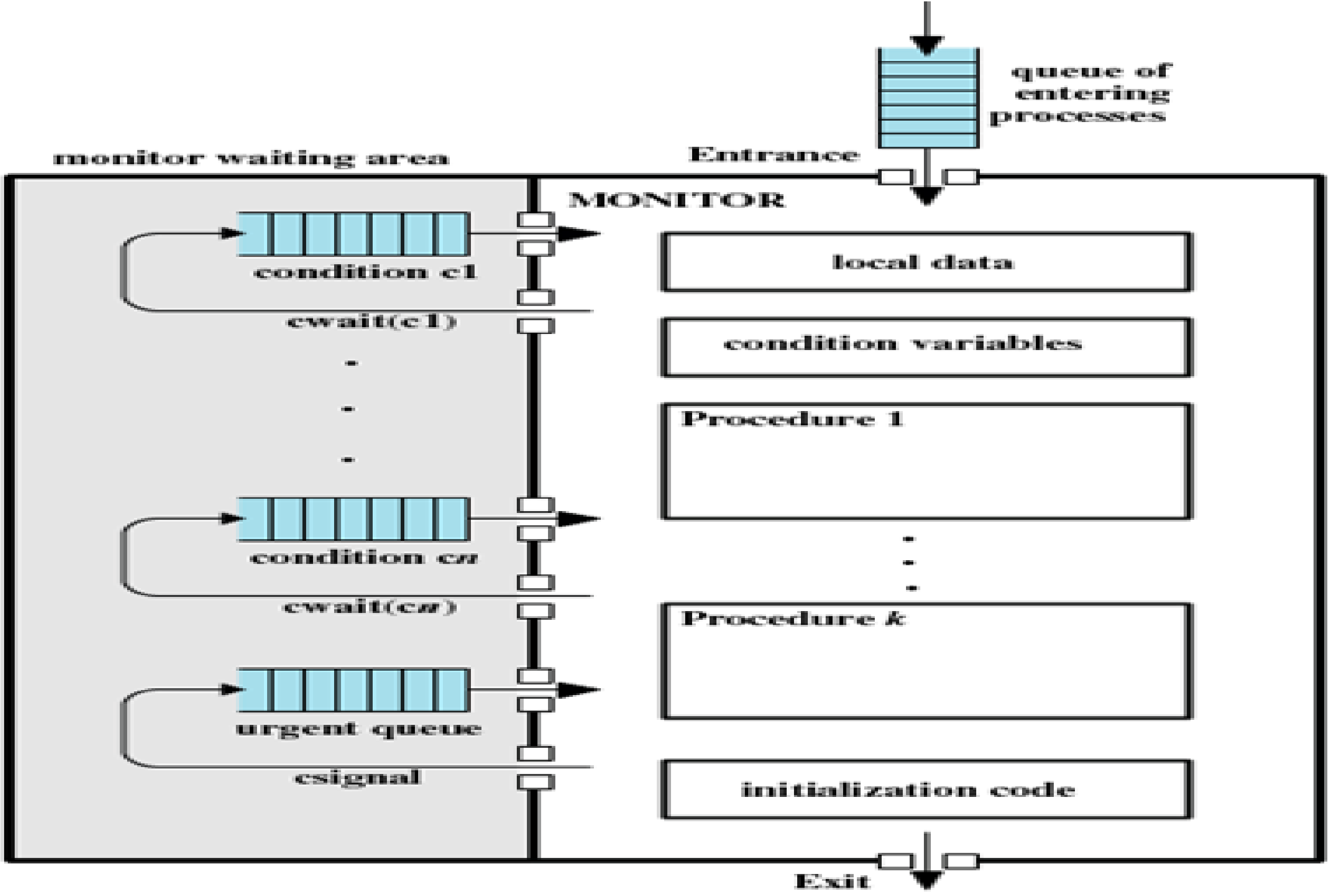


Condition variables

- To allow a process to wait within the monitor, a *condition* variable must be declared, as:
  - condition x, y;
- Two operations on a condition variable:
  - x.wait ()
    - a process that invokes the operation is suspended **until another process invokes x.signal**
  - x.signal ()
    - resumes exactly one suspended process**. If no process is suspended, then the signal operation has no effect.**

Monitor with Condition variables

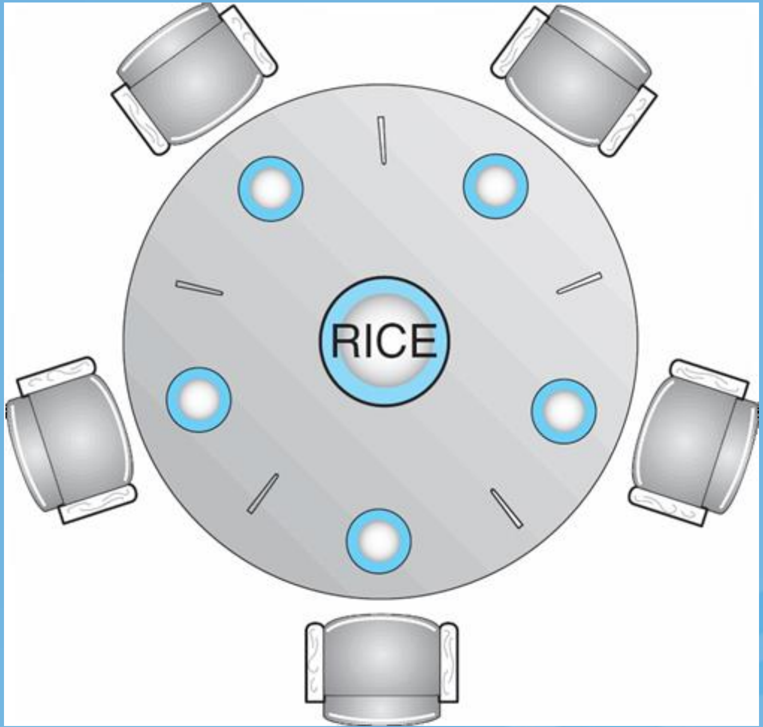



queue of entering processes
Entrance
monitor waiting area
MONITOR
condition c1
cwait(c1)
local data
condition variables
Procedure 1
condition cn
cwait(cn)
Procedure k
urgent queue
csignal
initialization code
Exit

Condition Variables Choices

- If process P invokes `x.signal()`, and process Q is suspended in `x.wait()`, what should happen next?
  - Both Q and P cannot execute in parallel. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed

Dining Philosopher Problem



Solution to Dining Philosopher Problem

```
monitor DP
  {
      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 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 () ;
   }
}

initialization_code()
{
        for (int i = 0; i < 5; i++)
        state[i] = THINKING;
}

}
```

Solution to Dining Philosopher Problem

- Each philosopher *I* invokes the operations pickup() and putdown() in the following sequence:

DiningPhilosophters.pickup (i);

EAT

DiningPhilosophers.putdown (i);

Monitor Implementation using Semaphores

- Variables

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

- Each 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.signal can be implemented as:

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

The operation x.wait can be implemented as:

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

Kernel Synchronization - Windows

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
  - Spinlocking-thread will never be preempted
- Also provides **dispatcher objects** user-land which may act mutexes, semaphores, events, and timers
  - **Events**
    - An event acts much like a condition variable
  - Timers notify one or more thread when time expired
  - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)

Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - Semaphores
  - atomic integers
  - spinlocks
  - reader-writer versions of both
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

Linux Synchronization

- Atomic variables

  `atomic_t` is the type for atomic integer
- Consider the variables

```
atomic_t counter;
int value;
```

Atomic Operation	*Effect*
`atomic_set(&counter,5);`	`counter = 5`
`atomic_add(10,&counter);`	`counter = counter + 10`
`atomic_sub(4,&counter);`	`counter = counter - 4`
`atomic_inc(&counter);`	`counter = counter + 1`
`value = atomic_read(&counter);`	`value = 12`

POSIX Synchronization

- POSIX API provides
  - mutex locks
  - semaphores
  - condition variable
- Widely used on UNIX, Linux, and macOS