SLIIT
Discover Your Future

IT2060/IE2061

Operating Systems and System Administration

Lecture 07

Process Syncronization

U. U. Samantha Rajapaksha

M.Sc.in IT, B.Sc.(Engineering) University of Moratuwa

Senior Lecturer SLIIT

Samantha.r@slit.lk

Chapter 5: Process Synchroniza[redacted]

- Background
- The Critical-Section Problem
- Semaphores
- Classic Problems of Synchronization
- Monitors

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:
  Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers. We can do so by having an integer **`counter`** that keeps track of the number of full buffers. Initially, **`counter`** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a 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 */
}
```

Race Condition

- **`counter++`** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **`counter--`** could be implemented as

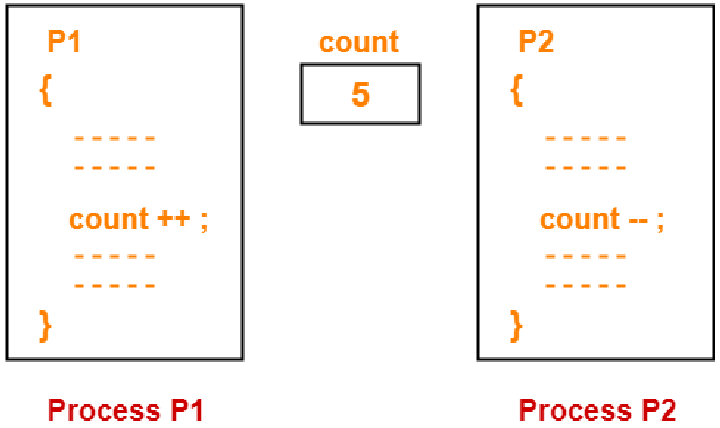
```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter` {register1 = 5}
S1: producer execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 – 1` {register2 = 4}
S4: producer execute `counter = register1` {counter = 6 }
S5: consumer execute `counter = register2` {counter = 4}

Race condition

- *Race condition* is a situation where several processes access and manipulate the same data concurrently.
  - The outcome of the execution depends on particular order in which the access takes place.
- In order to prevent race condition on *counter*, we need to ensure that only one process at a time can be manipulating *counter*
  - We need some form of *process synchronization*.



Critical Section Problem

- Consider system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structu

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

Solution to Critical-Section Prob[redacted]

1. **Mutual Exclusion** - If process ***P_i*** is executing in its critical section, then no other processes can b[redacted] executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

Solution (cont.)

Simplest solution:

Each process *disables* all interrupts just *after entering* its critical section and *re-enables* them just *before leaving* it.

- Not wise, because enabling/disabling interrupt is a privileged instruction.

Solution in kernel mode:

- **Preemptive kernel:** a process can be preempted while running in the kernel mode
  - Otherwise, the kernel is **nonpreemptive kernel**: allows the process to run until it exits kernel mode, blocks, or voluntarily yields CPU.
- Nonpreemptive kernel is free from race conditions on kernel data structure
  - However preemptive kernel is more responsive and suitable for real time system.

Solution for two processes, P_0 and P_1

Software-based

ALGORITHM 1

var *turn*: (0 .. 1); // Initially turn = 0; *turn* = *i* means P_i can enter its CS

Process P_i

repeat

while *turn* ≠ *i* **do** *no-op*;

Critical Section

turn = *j*;

Remainder Section

until *false*;

Process P_j

repeat

while *turn* ≠ *j* **do** *no-op*;

Critical Section

turn = *i*;

Remainder Section

until *false*;

- Satisfies mutual exclusion, but not progress requirement.
  - If *turn* = 0, P_1 cannot enter its CS even though P_0 is in its RS.
  - Taking turn is not good when one process is slower than other.
- *Busy waiting*: continuously testing a variable waiting for some value to appear
  - not good since it wastes CPU time

Solution for two processes (cont.)

Software-based

ALGORITHM 2

// Initially *flag*[0] = *flag*[1] = *false*; *flag[i]* = *true* means P_i wants to enter its CS

var *flag*: **array** [0 .. 1] **of** *boolean;*

Process P_i

repeat

    flag[*i*] = *true*;

    while *flag*[*j*] **do** *no-op***;**

        Critical Section;

    flag[*i*] = *false*;

        Remainder Section;

until *false*;

Process P_j

repeat

    flag[*j*] = *true*;

    while *flag*[*i*] **do** *no-op***;**

        Critical Section;

    flag[*j*] = *false*;

        Remainder Section;

until *false*;

- Satisfy mutual exclusion, but violates the progress requirement:
  T_0: P_0 sets flag[0] = true.
  T_1: P_1 sets flag[1] = true.
  → P_0 and P_1 are looping in their respective **while.**

Solution for two processes (cont.)

Software-based

// **Peterson's solution**: Combine shared variables of Algorithms 1 and 2

Process P_i

repeat
 flag[*i*] = *true*;
 turn = *j*;
 while (*flag*[*j*] **and** *turn* = *j*) **do** *no-op*;
  critical section
 flag[*i*] = *false*;
  remainder section
until *false*;

Process P_j

repeat
 flag[*j*] = *true*;
 turn = *i*;
 while (*flag*[*i*] **and** *turn* =*i*) **do** *no-op*;
  critical section
 flag[*j*] = *false*;
  remainder section
until *false*;

- Solves the critical-section problem for two processes.
  - It meets all the three requirements
- Proof: need to show that:
  - Mutual exclusion is preserved.
  - The progress requirement is satisfied.
  - The bounded waiting time requirement is met.
- For detailed proof, read the textbook.

Bakery Algorithm

- The solution to the critical section problem for *n* processes by Leslie Lamport
- Before entering its critical section, each process receives a number.
  - The holder of the smallest number enters the critical section.
  - If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first.
- Notation (**ticket#**, process **id#**)
  - $(a, b) < (c, d)$ if $a < c$ or if $a = c$ and $b < d$.
  - $max\ (a_0, \ldots, a_{n-1})$ is a number k such that $k \geq a_i$ for $i = 0, \ldots, n-1$.
- shared data
  - **var** *choosing*: **array**[0..*n*-1] of *boolean*;
  - *number*: **array** [0 .. *n*-1] of *integer*;
- data structures are initialised to *false* and 0, respectively

Bakery Algorithm (contd.)

Process P_i

```
repeat
    choosing[i] = true;
    number[i] = max (number[0], number[1], …, number[n-1]) +1;
    choosing[i] = false;
    for j = 0 to n-1 do
    begin
        while choosing[j] do no-op;
          while number[j] ≠ 0 and (number[j],  j) < (number[i], i) do no-op;
    end;
            critical section
    number[i] = 0;
            remainder section
until false
```

Synchronization Hardware

- There is no guarantee that the software-based solution will work correctly in all computer architectures.
- The simple solution to critical section problem: disable interrupt while a shared variable is being modified.
  - This solution is not feasible in multiprocessor. Why?
- Use special hardware instructions such as *Test-and-Set* and *Swap.*
  - *Test-and-set* or *Swap* is an atomic instruction: it can not be interrupted until it completes its execution

```
// Test and set the content of a
   word atomically

function Test-and-Set (var
   boolean: target)
begin
   Test-and-Set = target;
   target = true;
end;
```

```
// Swapping  instruction is
   done atomically

procedure Swap (var
   boolean: a, b)
var boolean: temp;
begin
   temp = a;
   a = b;
   b = temp;
end;
```

How to use them?

Mutual Exclusion with *Test-and-Set*

var *boolean*: *lock*; *lock* is a shared variable, initially set to *false*.

Repeat // *Process P_i*
  while *Test-and-Set* (*lock*) **do** *no-op*;
    Critical Section
  lock = *false*;
    Remainder Section
until *false*;

Repeat // *Process P_j*
  while *Test-and-Set* (*lock*) **do** *no-op*;
    Critical Section
  lock = *false*;
    Remainder Section
until *false*;

Mutual Exclusion with *Swap*

Repeat // *Process P_i*
  key = *true*;
  repeat
    Swap (*lock*, *key*);
  until *key* = *false*;
    Critical section
  lock = *false*;
    Remainder section
until *false*;

Repeat // *Process P_j*
  key = *true*;
  repeat
    Swap (*lock*, *key*);
  until *key* = *false*;
    Critical section
  lock = *false*;
    Remainder section
until *false*;

- Both do not satisfy the bounded waiting requirement.

SLIIT FACULTY OF COMPUTING

Correct solution with Test-and-set

Shared data: **var** *waiting*: **array**[0..*n*-1] **of** *boolean*; *lock*: *boolean*; //All initialized to *false*

Process P_i

```
var  j: 0..n-1;  key: boolean;
repeat
    waiting[i] = true;
    key = true;
    while waiting [i] and key do          // enter CS if either waiting[i] or key is false
            key = Test-and-Set (lock);     // key is false if lock is false
    waiting[i] = false;
            Critical Section
    j = i+1 mod n;
    while (j ≠ i) and not waiting[j] do  // check if any Pj is waiting for CS
            j = j+1 mod n
     if j = i then
            lock = false;                  // no other process is waiting for CS
     else
             waiting[j] = false;           // Pj is waiting, let it enter CS next

             Remainder Section
until false;
```

Proof: Read textbook.

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for p[redacted] synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`**
- Definition of the **`wait()` operation**

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

- Definition of the **`signal()` operation**

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

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

  Create a semaphore "**synch**" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

- Can implement a counting semaphore S as a binary semaphore

Semaphore Implementation

[redacted]

- Must guarantee that no two processes can execute[redacted] the `wait()` and `signal()` on the same semaphore at th[redacted] same time
- Thus, the implementation becomes the critical section problem where the `wait` and `signal` code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue
-
```
typedef struct{
int value;
struct process *list;
} semaphore;
```

Implementation with no Busy waiting (Cont.)

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

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

Classical Problems of Synchroni[redacted]

- Classical problems used to test new[redacted] proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- *n* buffers, each can hold one item
- Semaphore `mutex` initialized to the value 1
- Semaphore `full` initialized to the value 0
- Semaphore `empty` initialized to the value n

Bounded Buffer Problem (Cont.[redacted]

- The structure of the producer process

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

Bounded Buffer Problem (Cont[redacted]

- The structure of the consumer process

```
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

[redacted]

- A data set is shared among a number of concurren[redacted] processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore `rw_mutex` initialized to 1
  - Semaphore `mutex` initialized to 1
  - Integer `read_count` initialized to 0

Readers-Writers Problem (Cont.[redacted]

- The structure of a writer process

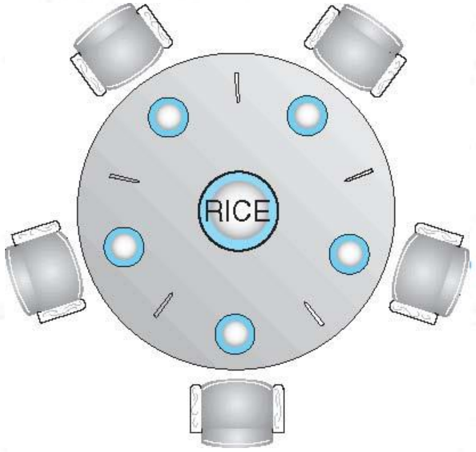
```
do {
     wait(rw_mutex);
        ...
     /* writing is performed */
        ...
   signal(rw_mutex);
} while (true);
```

Readers-Writers Problem (Cont.[redacted]

- The structure of a reader process

```
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



- Philosophers spend their lives alternating thinking and eating
- Don’t interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do {
     wait (chopstick[i] );
      wait (chopStick[ (i + 1) % 5] );

               //  eat

      signal (chopstick[i] );
      signal (chopstick[ (i + 1) % 5] );

                //  think

} while (TRUE);
```

- What is the problem with this algorithm?

Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section.
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation are possible.

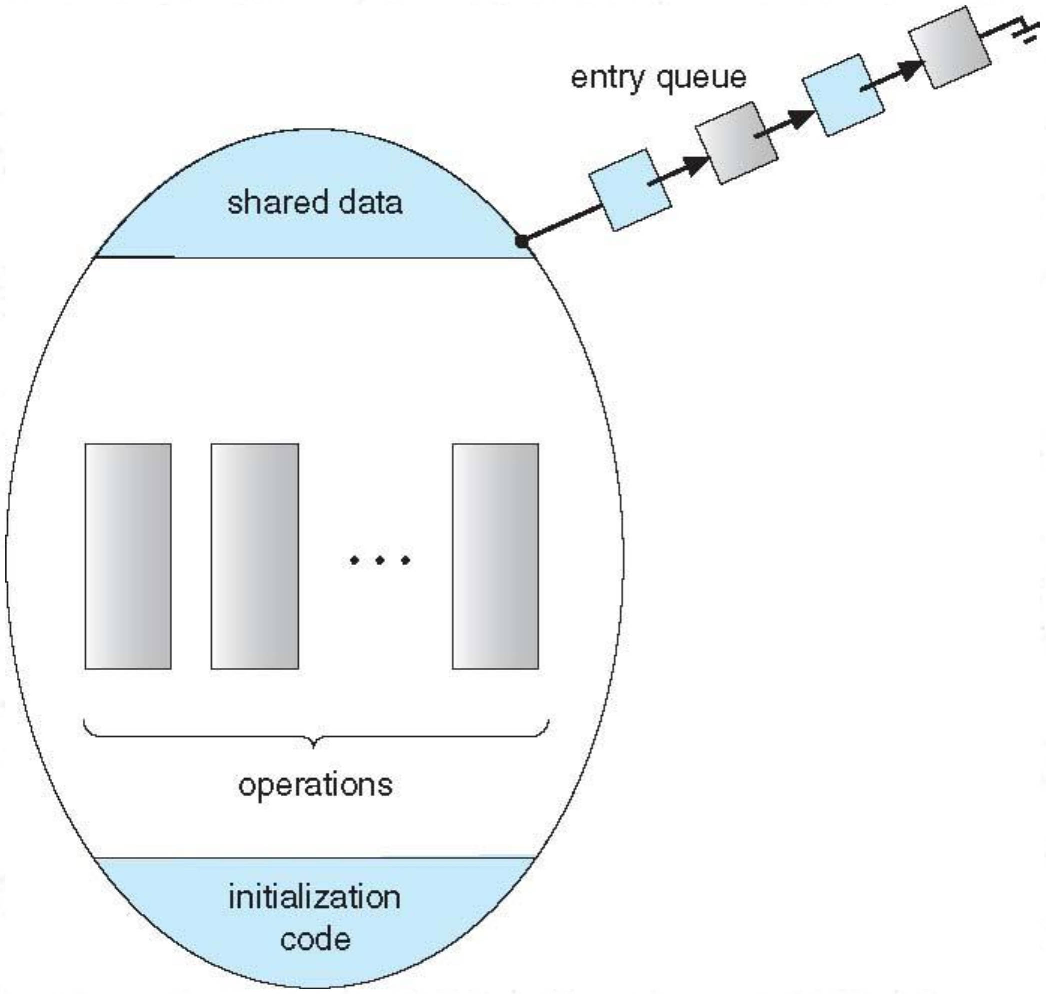
Monitors

[redacted]

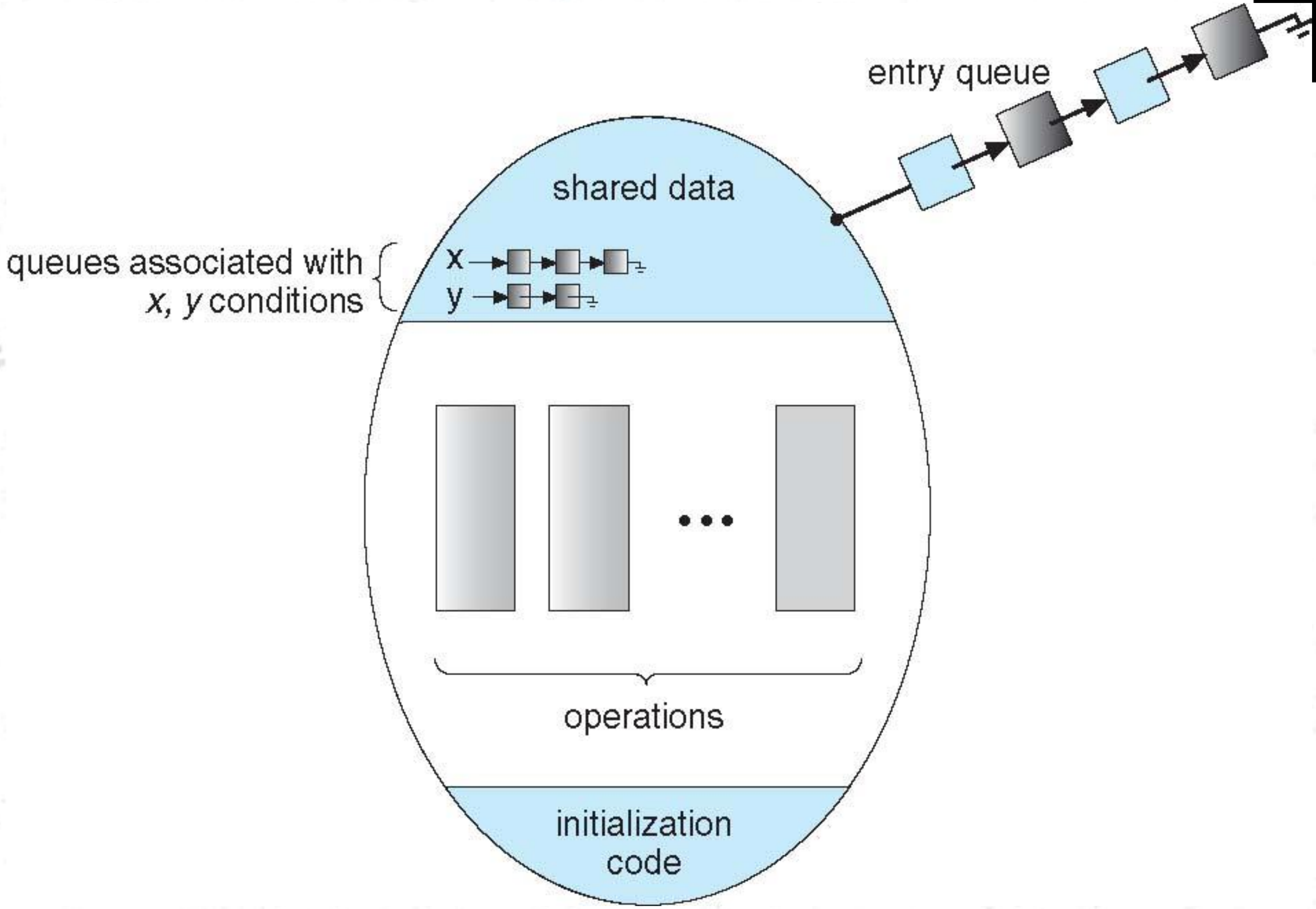
- A high-level abstraction that provides a convenient and effective mechanism fo[redacted] process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

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

  procedure Pn (…) {……}

     Initialization code (…) { … }
  }
}
```

Schematic view of a Monitor



Monitor with Condition Variable[redacted]



End of Chapter 5

U.U.Samantha Rajapaksha
Senior Lecturer
Coordinator-M.Sc. in IT
Faculty of Computing
B.Sc.(Engineering) Moratuwa, M.Sc. IT (SLIIT)
Email: samantha.r@sliit.lk
Tel:112301904 Ext:4116
Web:www.sliit.lk