Unit – 3

Part -2

Process Management
&
Synchronization

By

Dr Capt Ravindra Babu Kallam

Topics to be covered

- Critical Section Problem,
- Synchronization Hardware,
- Semaphores,
- Classical Problems of Synchronization
- Critical Regions
- Monitors

Dr R B Kallam

Critical Section Problem:

- Resources can be divided in to two classes
  - Consumable
  - Re usable
- These Resources can be either sharable or non sharable.
- In multiprocessing systems, the processes can compete for the resources and it leads to the *shared data problems.*
- *In the example cErrors is a common variable shared by two asynchronous processes simultaneously and led to shared data problem.*
- Such a variables / resources / etc are called critical resources and an area is called Critical Section and this situation is called *race condition*.

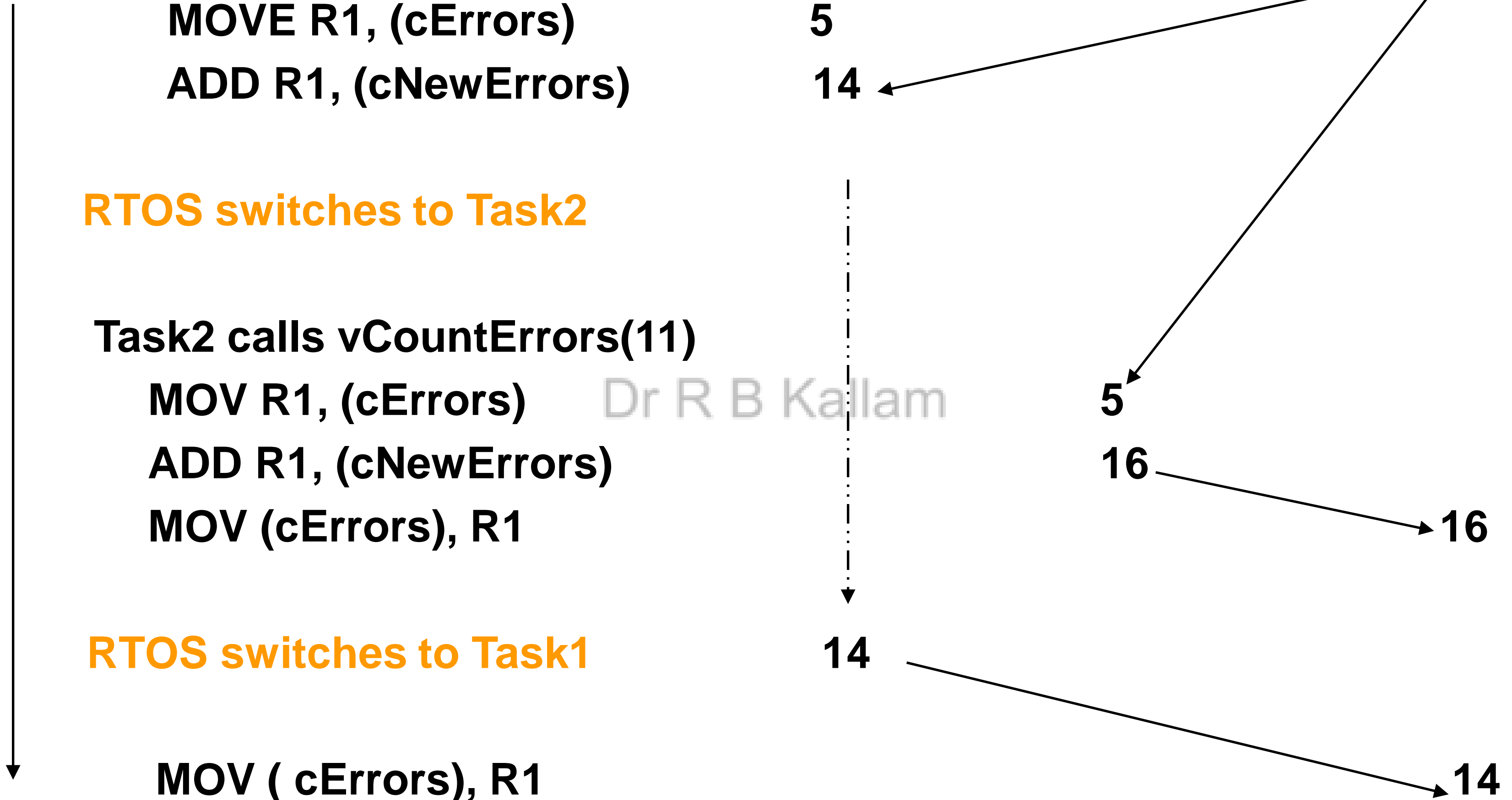
Dr R B Kallam

Example for Shared data Problem:

- **Assembly code for vcountErrors**

```
Void vCountErrors ( int cNewErrors)
{
   cErrors += cNewErrors;
    MOVE R1, (cErrors)
    ADD R1, ( cNewErrors)
    MOVE ( cErrors), R1
    RETURN
}
```

Dr R B Kallam



Dr R B Kallam

Producer-Consumer problem

- Let us consider of the bounded buffer, and at most BUFFER.SIZE - 1 items are allowed in the buffer at the same time.
- Add an integer variable counter, initialized to 0. counter is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer.

Dr R B Kallam

The code for the producer process is as follows:

```
while (true)
{
    /* produce an item in nextProduced */
    while (counter == BUFFER_SIZE)
      ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

The code for the consumer process is as follows:

```
while (true)
{
    while (counter == 0)
      ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    /* consume the item in nextConsumed */
}
```

- Although both the producer and consumer routines are correct separately, they may not function correctly when executed concurrently.
- As an illustration, suppose that the value of the variable counter is currently 5 and that the producer and consumer processes execute the statements "counter++" and "counter—" concurrently.
- Following the execution of these two statements, the value of the variable counter may be 4, 5, or 6! The only correct result, though, is counter == 5, which is generated correctly if the producer and consumer execute separately.
- We can show that the value of counter may be incorrect as follows.
- Note that the statement "counter++" may be implemented in machine language as follows; where register1 is a local CPU register.

$register_1$ = counter
$register_1$ = $register_1 + 1$
counter = $registeri$

- Similarly, the statement "counter --" is implemented as follows; where register2 is a local CPU register.

$register_2$ = counter
$register_2$ = $register_2 - 1$
counter = $registeri$

Dr R B Kallam

- Even though register1 and register2 may be the same physical register (an accumulator, say), the contents of this register will be saved and restored by the interrupt handler.
- The concurrent execution of "counter++" and "counter—" is interleaved as shown below:

T_0:	*producer*	execute	*registeri* — counter	{$register_1$ = 5}
T_1:	*producer*	execute	$register_1 = register_1 + 1$	{$register_1$ = 6}
Tr.	*consumer*	execute	$register_2$ = counter	{register2 = 5}
T_3:	*consumer*	execute	$register_2$ = *registeri* — 1	{$register_2$ = 4}
T_4:	*producer*	execute	counter = *registeri*	{counter = 6}
T_5:	*consumer*	execute	counter = $register_2$	{counter = 4}

Dr R B Kallam

- Notice that we have arrived at the incorrect state "counter == 4", indicating that four buffers are full, when, in fact, five buffers are full.
- We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently. This situation is called race condition. The variable Counter is called critical resource and an area is called Critical Section.

A solution to the critical-section problem must satisfy the following three requirements:

1. **Mutual exclusion:** If process P; is executing in its critical section, then no other processes can be executing in their critical sections.

2. **Progress:** If no process is executing in its critical section and some processes wish to enter their critical sections, must be permitted without any delay.

Dr R B Kallam

3. **Bounded waiting:** There exists a bound, or limit, 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.

Mutual Exclusion Requirements:

- Only one process at a time is allowed into its critical section among all processes that need to access critical section
- A process that halts in its non critical section must do so without interfering with other processes
- It must not be possible for processes requiring access to a critical section to be delayed indefinitely; no deadlock or starvation can be allowed.
- When no process is in critical section, any process that request entry to its critical section must be permitted to enter without delay.
- A process remains inside critical section for a finite time only.

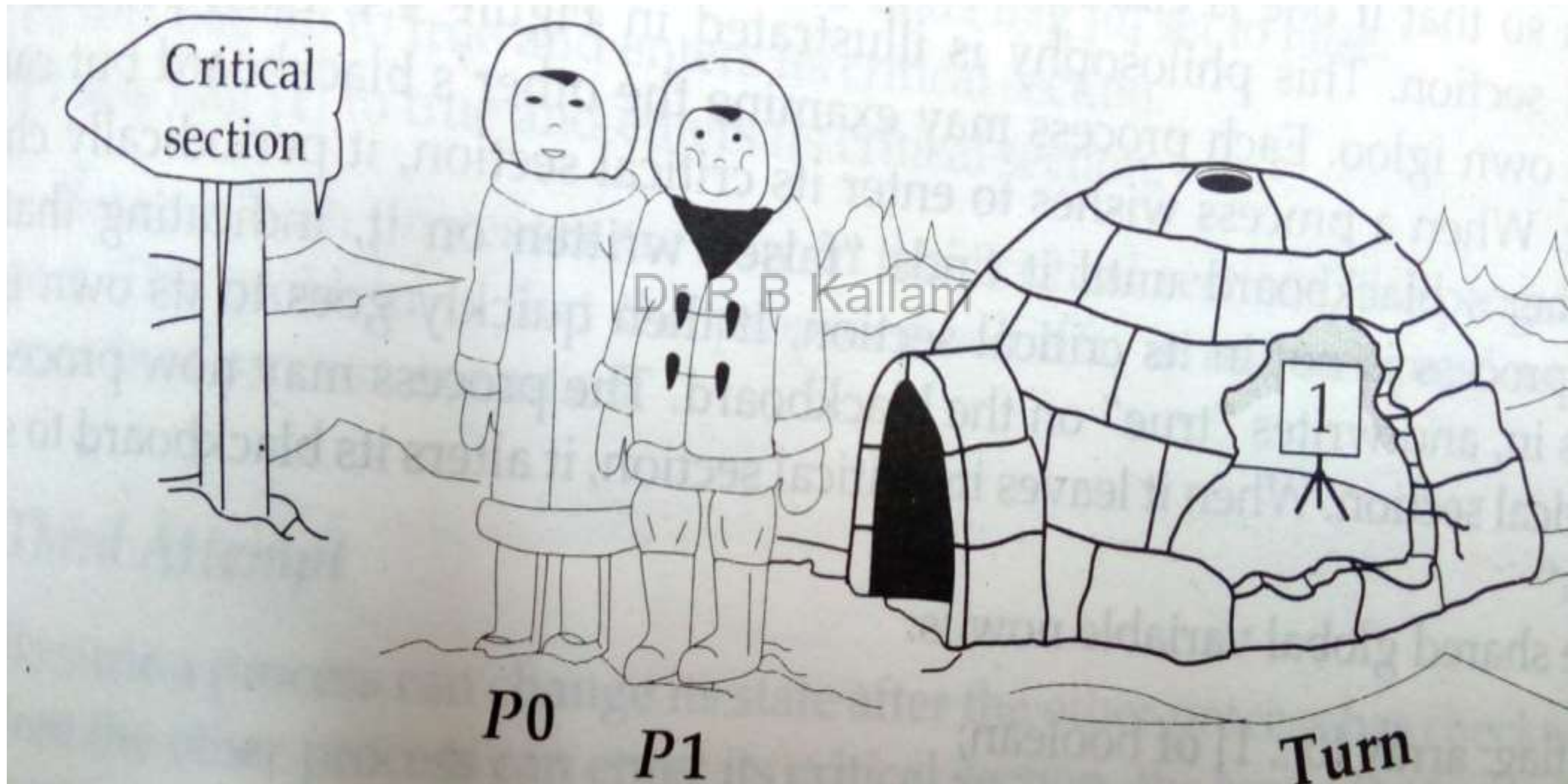
Dr R B Kallam

- For Mutual exclusion we have two approaches:
  - Using Software Approach
  - Using Hardware Approach

Dr R B Kallam

Two-Process Solutions Using Dekker's Algorithm- Software approach

First Attempt



- A Process (P0 or P1) that wish to execute its critical section first enters the igloo and examines the black board. If its number is on the black board, then the process may leave the igloo and proceeds to its critical section, otherwise, it leaves the igloo and is forced to wait.
- From time to time the process reenters the igloo to check the blackboard. It repeats this exercise until it is allowed to enter its critical section. This procedure is known as busy-waiting.
- After a process has gained access to its critical section and after its has completed that section, it must return to the igloo and place the number of the other process on the board.

Dr R B Kallam

Drawbacks:

- If one process fails the other process is permanently blocked.
- If one process (P0) needs to enter its critical section once in a Hour and the other (P1) wants to enter 100times in an hour , then P1 has to Busy-wait until P0 changes the number board.

Dr R.B Kallam

- **Algorithm 1**
  - Assume the two processes as P0and P1
  - Let the process share a common integer variable turn initialized to 0(or 1).

Process 0

Dr R B Kallam

.

.

While turn ≠ 0 do no-op;

<critical section>

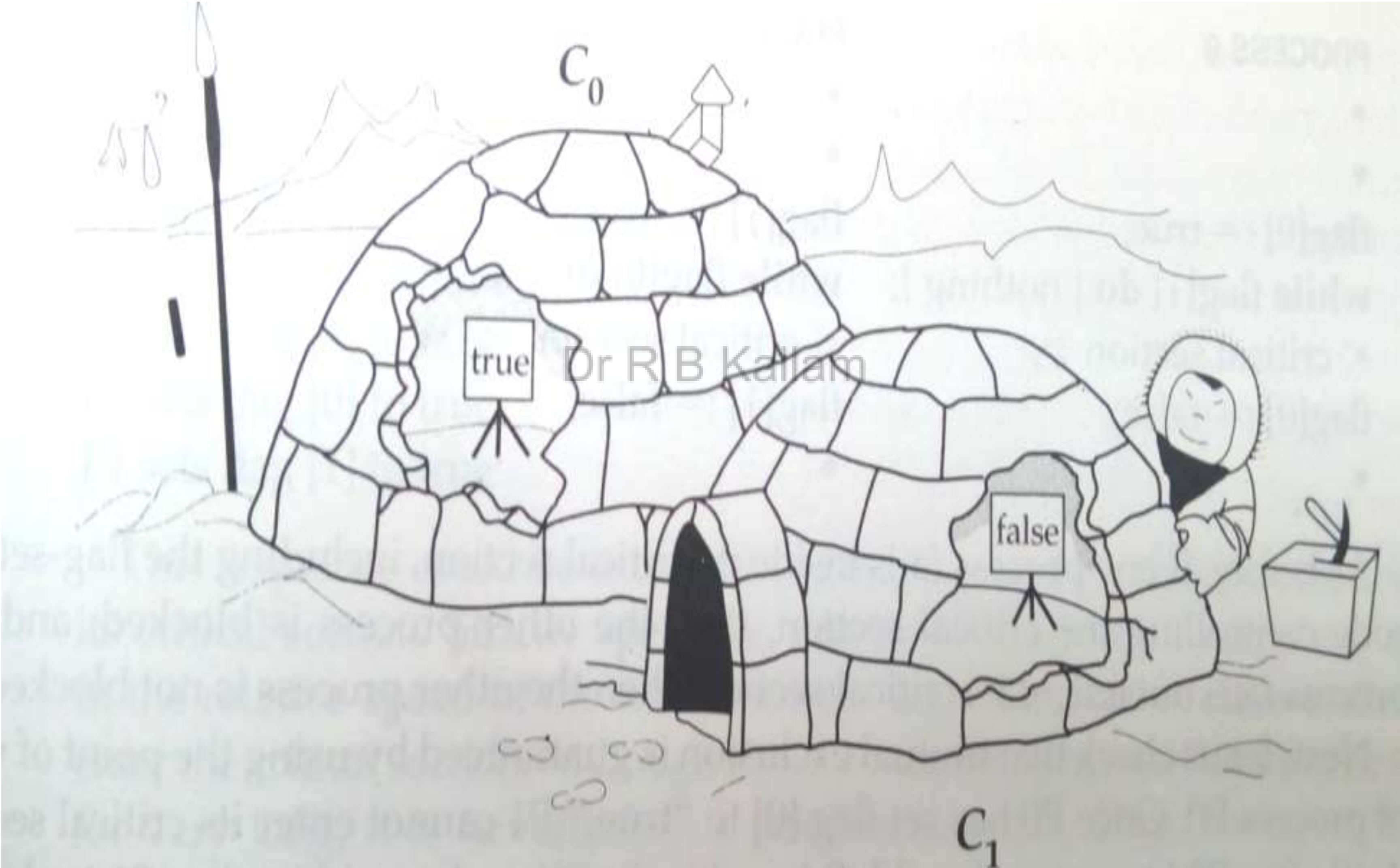
turn:= 1;

remainder section

.

.

Second Attempt



- **Algorithm 2**

repeat

while flag [1] do no-op;
flag [0] := true;

<critical section>
flag [0] := false;

Dr R B Kallam

remainder section

until ***false*****;**

The structure of process P0 in algorithm 2

- Algorithm 3

```
repeat
        flag [0] := true;
        while flag [1] do no-op;

        <critical section>

        flag [0] := false;

        remainder section

until false;
```

Dr R B Kallam

The structure of process P0 in algorithm 3

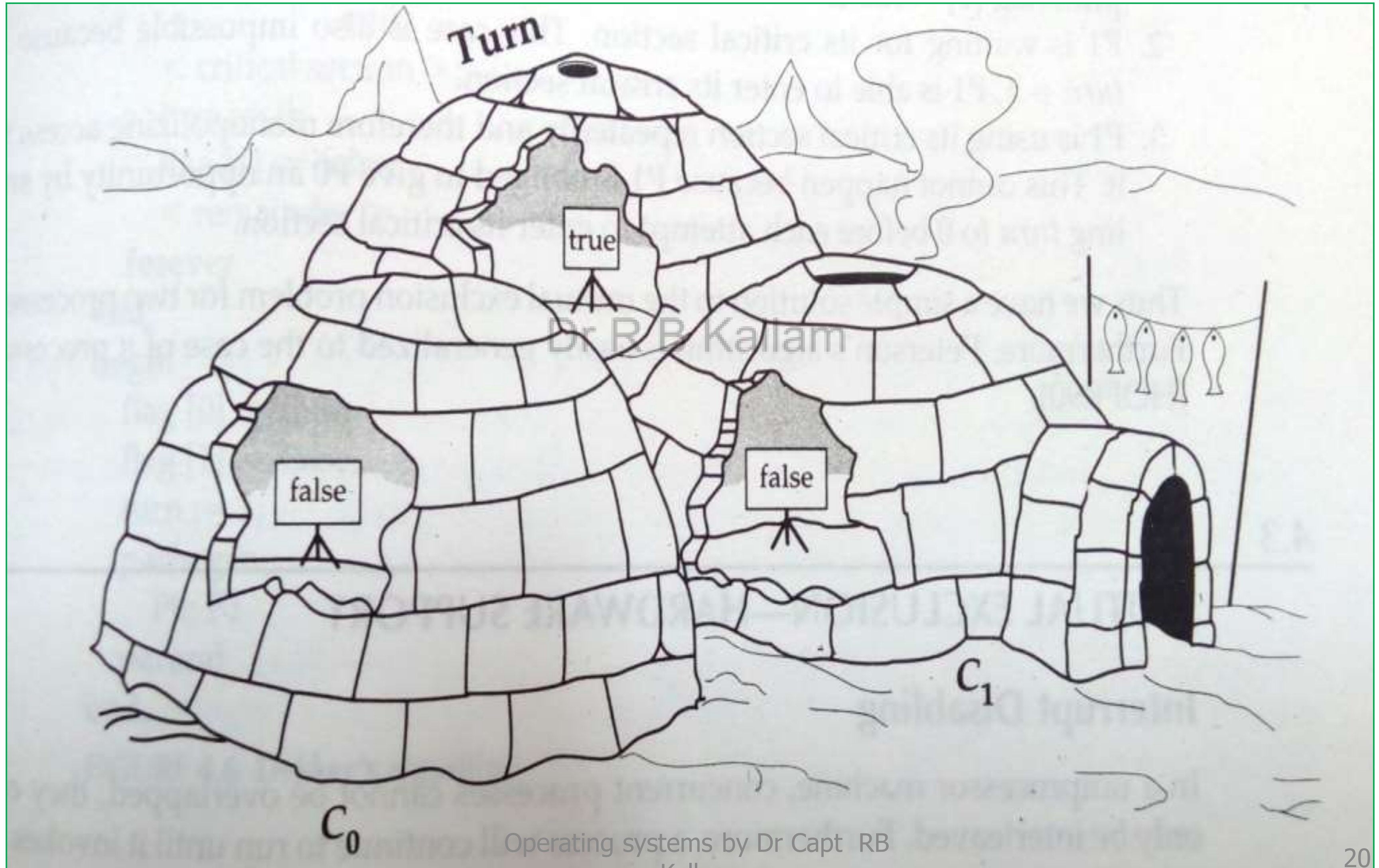
- **Algorithm 4**

```
repeat
        flag [0] := true;
        while flag [1] do no-op;
        begin
                flag [0] := false;
                <delay for a short time>
                flag [0] := true;
        end;
        <critical section>

        flag [0] := false;
        remainder section

until false;
```

The structure of process P0 in algorithm 4

Dr R B Kallam

A Correct Solution



- There is a referee igloo with a black board labeled “turn”.
- When P0 wants to enter the critical section, it sets the flag to “true”.
- It then goes and check the flag of P1. If that is “false”, P0 may immediately enter the critical section.
- Otherwise, P0 goes to consult the referee. If it finds that turn=0, then it knows that it is its turn to insist, and periodically checks P1 igloo.
- P1 will at some point will change its blackboard to “false”, allowing P0 to proceed.
- After P0 has used the critical section, it sets the flag to “false” to free the critical section and sets “turn” to 1 to transfer the right to insist to P1.

Dr R B Kallam

- **A Correct Solution by Dekker's**

```
repeat

        flag [0] := true;
        while flag[1] do if turn=1 then
                begin
                        flag [ 0 ] := false;
                        while ( turn = 1 ) do no-op;
                        flag [0] := true;
                end;
         <critical section>
        turn := 1;
        flag [ 0 ] := false;

        remainder section

        …….
```

Dr R B Kallam

The structure of process P0 in final algorithm

Peterson's solution

- A classic software-based solution to the critical-section problem known as Peterson's solution.
- Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered Po and Pi.
- Peterson's solution requires two data items to be shared between the two processes:
  - int turn;
  - boolean flag [2]

Dr R B Kallam

- The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i, then process P_i; is allowed to execute in its critical section.
- The flag array is used to indicate if a process is ready to enter its critical section. For example, if flag[i] is true, this value indicates that P_i; is ready to enter its critical section.
- Peterson preserves the following:
  - 1. Mutual exclusion .
  - 2. The progress requirement.
  - 3. The bounded-waiting requirement.

```
do {

    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

Figure 6.2 The structure of process P_i in Peterson's solution.

Dr R B Kallam

```
boolean flag [2];
int turn;
void P0()
{
      while (true) {
            flag [0] = true;
            turn = 1;
            while (flag [1] && turn == 1) /* do nothing */;
            /* critical section */;
            flag [0] = false;
            /* remainder */;
      }
}
void P1()
{
      while (true) {
            flag [1] = true;
            turn = 0;
            while (flag [0] && turn == 0) /* do nothing */;
            /* critical section */;
            flag [1] = false;
            /* remainder */
      }
}
void main()
{
      flag [0] = false;
      flag [1] = false;
      parbegin (P0, P1);
}
```

Dr R B Kallam

Figure: Peterson's Algorithm for Two Processes

Synchronization Hardware:

Interrupt Disabling:

- The critical-section problem in uniprocessor systems can be solved by preventing interrupts from occurring while a shared variable was being modified.
- With this approach, the current sequence of instructions would be allowed to execute in order without preemption. This is the approach taken by nonpreemptive kernels.

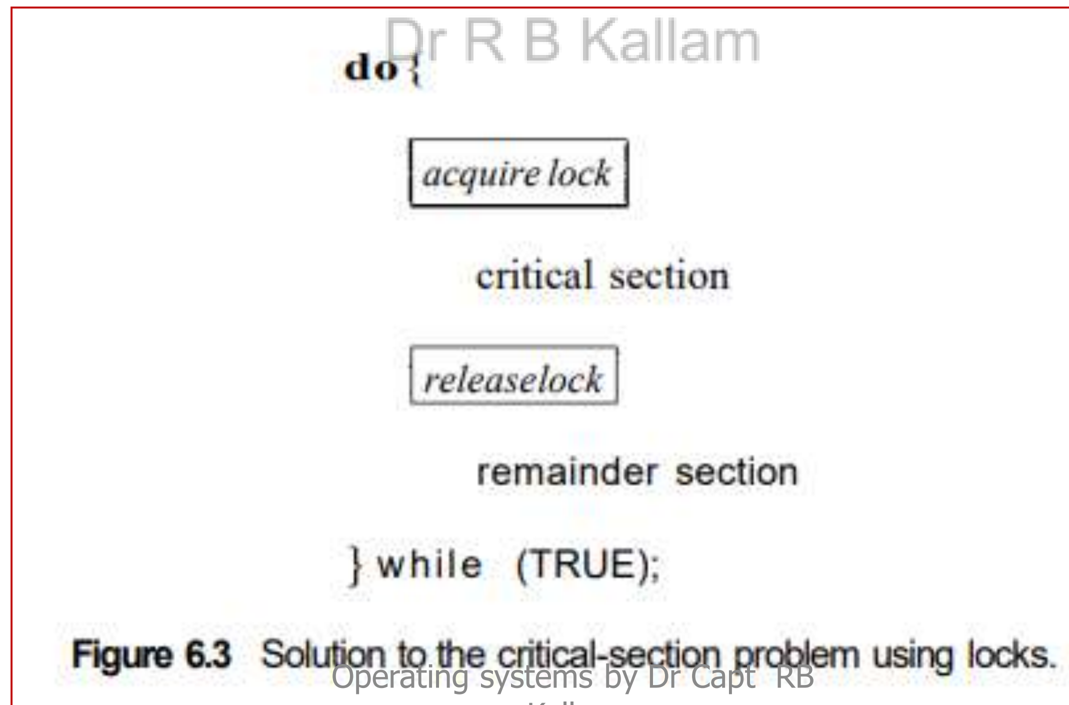
Dr R B Kallam

- This solution is not as feasible in a multiprocessor environment.
- Disabling interrupts on a multiprocessor can be time consuming, as the message need to be passed to all the processors; system efficiency decreases.

```
repeat
        < disable interrupts>;
        <critical section>;
        <enable interrupts>;
        <remainder section>;
forever.
```

Special Machine Instructions:

- In general, we can state that any solution to the critical-section problem requires a simple tool—a lock.
- Race conditions are prevented by requiring that critical regions be protected by locks.
- That is, a process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section.



Dr R B Kallam

Figure 6.3 Solution to the critical-section problem using locks.

TestAndSet() Instruction:

- Mutual exclusion can be achieved by using a special machine instruction called TestAndSet() .
- The important characteristic is that this instruction is executed atomically.
- If the machine supports the TestAndSet () instruction, then we can implement mutual exclusion by declaring a Boolean variable lock, initialized to False.
- The structure of process P, is shown below:

Dr R B Kallam

```
do {
     while (TestAndSet (&lock)); // do nothing

          lock = TRUE;

          < critical section >

          lock = FALSE;

          <remainder section>

   } while(TRUE);
```

Figure : Mutual-exclusion implementation with TestAndSet ().

Swap() instruction:

- The Swap() instruction, operates on the contents of two words; it is defined as shown in Figure 6.6. It is executed atomically.
- If the machine supports the Swap() instruction, then mutual exclusion can be provided as follows:
- A global Boolean variable lock is declared and is initialized to false. (lock=False)
- In addition, each process has a local Boolean variable key.
- The structure of process P, is shown in Figure 6.7.
- Although these algorithms satisfy the mutual-exclusion requirement, they do not satisfy the bounded-waiting requirement.

Dr R B Kallam

```
void Swap(boolean *a, boolean *b) {
   boolean temp = *a;
   *a = *b;
   *b = temp;
}
```

Figure 6.6 The definition of the Swap () instruction.

```
do {
   key = TRUE;
   while (key == TRUE)
      Swap(&lock,&key);

      // critical section

   lock = FALSE;

      // remainder section
}while (TRUE);
```

Figure 6.7 Mutual-exclusion implementation with the Swap() instruction.

Properties of the Machine-Instruction Approach:

- **Advantages:**
  - It is applicable to any number of processes on either a single processor or multiple processors sharing main memory.
  - It is simple and therefore easy to verify.
  - It can be used to support multiple critical sections.
- **Disadvantages:**
  - **Busy – waiting is employed:** while a process is waiting for access to a critical section, it continues to consume processor time
  - **Starvation is possible:** When a process leaves a critical section and more than one process is waiting, the selection of a waiting process is arbitrary. Thus, some process could indefinitely be denied access.
  - **Dead lock is possible:** Consider the following scenario on a single-processor system.
    - Process P1 executes the special instruction (e.g., swap, exchange) and enters its critical section.
    - P1 is then interrupted to give the processor to P2, which has higher priority.
    - If P2 now attempts to use the same resource as P1, it will be denied access because of the mutual exclusion mechanism.
    - Thus, it will go into a busy waiting loop. However, P1 will never be dispatched because it is of lower priority than another ready process, P2.

Dr R B Kallam

Semaphores:

- **Semaphores are used to solve critical section problems. This is a synchronization tool.**
- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait () and signal ().
- **The following three operations are defined on Semaphore (S)**
  - **A semaphore may be initialized to a non - negative value.**
  - **The wait operation decrements the semaphore value. If the value become negative, then the process executing the wait is blocked.**

    Wait (s): While S < 0 do no – op;

    S:= S- 1;
  - **The signal operation increments the semaphore value. If the value is less then are equal zero, then a process blocked by a wait operation is unblocked.**

    Signal (S): S:= S+1;

Dr R B Kallam

Mutual exclusion using semaphores

```
do{
        wait(s);
         <critical section>
        signal (s);
         < remainder section>

    } while(true)
```

Dr R B Kallam

Semaphore operations

```
struct semaphore {
        int count;
        queueType queue;
};
void semWait(semaphore s)
{
        s.count--;
        if (s.count < 0) {
          /* place this process in s.queue */;
          /* block this process */;
        }
}
void semSignal(semaphore s)
{
        s.count++;
        if (s.count<= 0) {
          /* remove a process P from s.queue */;
          /* place process P on ready list */;
        }
}
```

Dr R B Kallam

Semaphore variants:

- Binary Semaphores: It is with an integer value that can range only between 0 and 1. It is simple to implement then the other semaphores.
- Mutex Semaphore: Similar to a binary semaphore. A key difference between the two is that the process that locks the mutex (sets the value to zero) must be the one to unlock it. these are specially used to solve priority inversion problem.
- Counting Semaphores: Some semaphores that can be taken multiple times are called counting semaphores.

Dr R B Kallam

Binary Semaphore

A binary semaphore may only take on the values 0 and 1 and can be defined by the following three operations:

1. A binary semaphore may be initialized to 0 or 1.
2. The semWaitB operation checks the semaphore value. If the value is zero, then the process executing the semWaitB is blocked. If the value is one, then the value is changed to zero and the process continues execution.
3. The semSignalB operation checks to see if any processes are blocked on this semaphore (semaphore value equals 0). If so, then a process blocked by a semWaitB operation is unblocked. If no processes are blocked, then the value of the semaphore is set to one.

Dr R B Kallam

Binary Semaphore

```
struct binary_semaphore {
        enum {zero, one} value;
        queueType queue;
};
void semWaitB(binary_semaphore s)
{
        if (s.value == one)
             s.value = zero;
        else {
                     /* place this process in s.queue */;
                     /* block this process */;
        }
}
void semSignalB(semaphore s)
{
        if (s.queue is empty())
             s.value = one;
        else {
                     /* remove a process P from s.queue */;
                     /* place process P on ready list */;
        }
}
```

Dr.R.B.Kallam

Figure : A Definition of Binary Semaphore Primitives

Mutex Semaphore

- Consider n processes, identified in the array P (i), all of which need access to the same resource.
- Each process has a critical section used to access the resource.
- In each process, a semWait(s) is executed just before its critical section.
- If the value of s becomes negative, the process is blocked.
- If the value is 1, then it is decremented to 0 and the process immediately enters its critical section; because s is no longer positive, no other process will be able to enter its critical section.
- The semaphore is initialized to 1. Thus, the first process that executes a semWait will be able to enter the critical section immediately, setting the value of s to 0.
- Any other process attempting to enter the critical section will find it busy and will be blocked, setting the value of s to –1.
- Any number of processes may attempt entry; each such unsuccessful attempt results in a further decrement of the value of s .

Dr R B Kallam

- When the process that initially entered its critical section departs, s is incremented and one of the blocked processes (if any) is removed from the queue of blocked processes associated with the semaphore and put in a Ready state.
- When it is next scheduled by the OS, it may enter the critical section.

```
/* program mutualexclusion */
const int n = /* number of processes */;
semaphore s = 1;
void P(int i)
{
        while (true) {
                semWait(s);
                /* critical section */;
                semSignal(s);
                /* remainder */;
        }
}
void main()
{
        parbegin (P(1), P(2),..., P(n));
}
```

Dr R B Kallam

Figure: Mutual Exclusion Using Semaphores

Counting Semaphores:

- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.
- The semaphore is initialized to the number of resources available.
- Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count).
- When a process releases a resource, it performs a signal () operation (incrementing the count).
- When the count for the semaphore goes to 0, all resources are being used.
- After that, processes that wish to use a resource will block until the count becomes greater than 0.

Dr R B Kallam

Difference between Strong & weak semaphore:

- For both counting semaphores and binary semaphores, a queue is used to hold processes waiting on the semaphore.
- The question arises of the order in which processes are removed from such a queue.
- The fairest removal policy is first-in-first-out (FIFO): The process that has been blocked the longest is released from the queue first; a semaphore whose definition includes this policy is called a strong semaphore .
- A semaphore that does not specify the order in which processes are removed from the queue is a weak semaphore .

Dr R B Kallam

Problems with the Semaphore:

- Forgetting to take the Semaphore
- Forgetting to release the Semaphore
- Taking the wrong Semaphore
- Holding a Semaphore for too long.
- Deadlock and starvation
- Priority inversion and Priority inheritance

Dr R B Kallam

Note: The serious problem with the semaphore is that wait and signal operations may be scattered throughout a program, and it is not easy to see the overall effect of these operations on system.

Classical Problem of Synchronization:

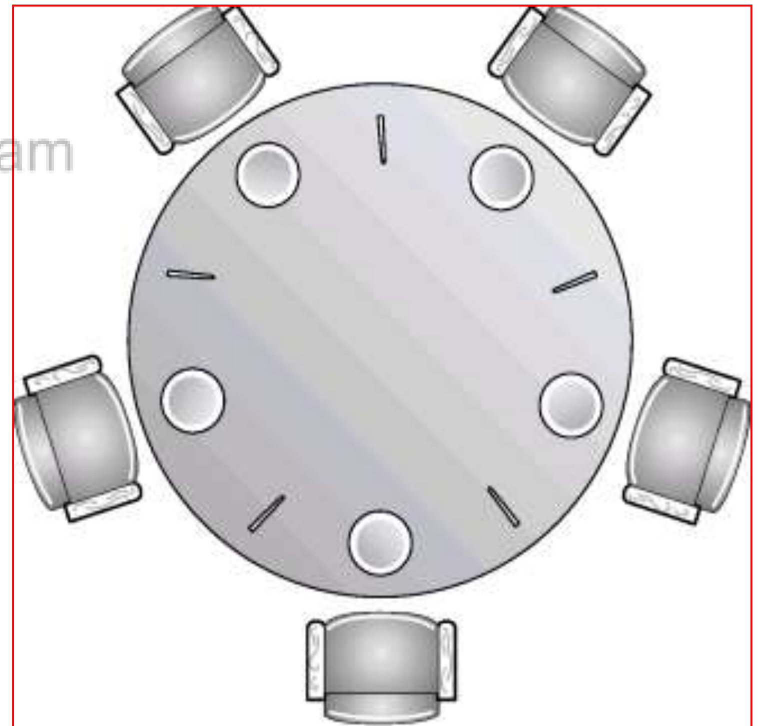
- Dining philosophers problem
- Reader writer problem
- The bounded buffer problem (or) producer – consumer problem

Dr R B Kallam

Dining philosophers problem

- Consider five philosophers who spend their lives thinking and eating.
- The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks (Figure 6.14).
- When a philosopher gets hungry, he tries to pick up the two chopsticks that are closest to him.
- A philosopher may pick up only one chopstick at a time. Obviously, he cannot pick up a chopstick that is already in the hand of a neighbor.
- When a hungry philosopher has both his chopsticks at the same time, he eats without releasing his chopsticks.
- When he is finished eating, he puts down both of his chopsticks and starts thinking again.

Dr R B Kallam



- One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait () operation on that semaphore; she releases her chopsticks by executing the signal() operation on the appropriate semaphores.

semaphore chopstick[5];

- Thus, the shared data are, where all the elements of chopstick are initialized to 1. The structure of philosopher / is shown in Figure 6.15. Although this solution guarantees that no two neighbors are eating simultaneously, it could create a deadlock.
- Suppose that all five philosophers become hungry and each grabs her left chopstick. All the elements of chopstick will now be equal to 0.
- When each philosopher tries to grab her right chopstick, she will be delayed forever and leads to deadlock.

Dr R B Kallam

Solution to the dining-philosophers problem that ensures freedom from deadlocks are:

- Allow at most four philosophers to be sitting simultaneously at the table
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this she must pick them up in a critical section).
- Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

Dr R B Kallam

```
do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   // eat
      . . .
   signal(chopstick[i]);

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

   // think

}while (TRUE);
```

Figure 6.15 The structure of philosopher *i*.

Reader writer problem

- There is a data area shared among a number of processes. The data area could be a file or a block of main memory, etc. There are a number of processes that only read the data area (readers) and a number that only write to the data area (writers).
- The conditions that must be satisfied are as follows:

  1. Any number of readers may simultaneously read the file.
  2. Only one writer at a time may write to the file.
  3. If a writer is writing to the file, no reader may read it.

Dr R B Kallam

- Thus, readers are processes that are not required to exclude one another and writers are processes that are required to exclude all other processes, readers and writers alike.
- In this readers do not write to the data area, nor do writers read the data area while writing.

Shared data

- *Integer readcount* initialized to 0;
- Var Semaphore mutex initialized to 1
- Var Semaphore wrt initialized to 1

Structure of writer process

```
While (true)
{
        wait (wrt);
        …
        //writing is performed//
        …
    signal(wrt);
};
```

Structure of Reader process

Mutex=1
readcount=0
wrt=1

```
While(true)
{
        wait(mutex);
        readcount := readcount +1;
        if (readcount == 1)  then wait(wrt);
            signal(mutex);
                        …
        // reading is performed in the critical section//
                        …..
        wait(mutex);
            readcount := readcount – 1;
        if (readcount == 0) then signal(wrt);
                        signal(mutex):
};
```

Dr R B Kallam

The bounded buffer problem (or) producer – consumer problem

- We assume that the pool consists of n buffers, each capable of holding one item.
- The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1.
- The empty and full semaphores count the number of empty and full buffers.
  - The semaphore empty is initialized to the value n
  - The semaphore full is initialized to the value 0.
  - Producer tries to insert data into an empty slot of the buffer
  - The consumer tries to remove data from a filled slot in the buffer
  - Producer must not insert data when the buffer is full
  - The consumer must not remove data when the buffer is empty.
- The general structure (code) for the producer process is shown in Figure 6.10; the code for the consumer process is shown in Figure 6.11.
- We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

Dr R B Kallam

```
do {
    . . .
  // produce an item in nextp
    . . .
  wait(empty);  //wait until empty>0
  wait(mutex);  //acquire lock
    . . .
  // add nextp to buffer
    . . .
  signal(mutex);  //release lock
  signal(full);   //increment full
}while (TRUE) ,-
```

Figure 6.10 The structure of the producer process.

```
do {
  wait(full);    //wait until full>0
  wait(mutex);   //acquire lock
    . . .
  // remove an item from buffer to nextc
    . . .
  signal(mutex);  //release lock
  signal(empty);  //increment empty
    . . .
  // consume the item in nextc
    . . .
}while (TRUE);
```

Figure 6.11 The structure of the consumer process.

Dr R B Kallam

Critical Regions

- Although semaphores provide a convenient and effective mechanism for process synchronization, their incorrect use can still result in timing errors that are difficult to detect, since these errors happen only if some particular execution sequences take place, and these sequences do not always occur.
- We have seen an example of such types of errors in the use of counters in shared data problems / producer consumer problem
- It is for this reason that the semaphores were introduced in the first place.
- Unfortunately, such timing errors can still occur with the use of semaphores.
- To illustrate how, let us review the solution to the critical- section problem using semaphores.
- All processes share a semaphore variable mutex, which is initialized to 1. Each process must execute wait(mutex) before entering the critical section, and signal(mutex) afterward.

Dr R B Kallam

- If this sequence is not observed, two processes may be in their critical sections simultaneously.
- Let us examine the various difficulties that may result. Note that these difficulties will arise even if a single process is not well behaved. This situation may be the result of an honest programming error or of an uncooperative programmer.
- Suppose that a process interchanges the order in which the wait and signal operations on the semaphore mutex are executed or taking the wrong semaphore or not considering the semaphore resulting in the following execution:
- These examples illustrate that various types of errors can be generated easily when semaphores are used incorrectly to solve the critical-section problem.

Dr R B Kallam

```
signal (mutex);
......................
<Critical section>
......................
Wait(mutex);
```

* Leads to failure of mutual exclusion

```
wait (mutex);
......................
<Critical section>
......................
Wait(mutex);
```

* Leads to the deadlock

```
......................
<Critical section>
......................
```

*either mutual exclusion is violated or a deadlock will occur

- To deal with the type of errors we have, a number of high-level language constructs, two of them are:
  - the critical region (sometimes referred to as conditional critical region).
  - the monitor.
- we assume that a process consists of some local data, and a sequential program that can operate on the data. *The local data can be accessed by only the sequential program that is encapsulated within the same process.*
- That is, one process cannot directly access the local data of another process. Processes can, however, share global data.
- The critical-region synchronization construct requires that a variable v of type T, which is to be shared among many processes, be declared as

var v: shared T;

- The variable v can be accessed only inside a region statement of the following form:

region v when B do S

Dr R B Kallam

- This construct means that, while statement S is being executed, no other process can access the variable v.
- The expression B is a Boolean expression that governs the access to the critical region. When a process tries to enter the critical section region, the Boolean expression B is evaluated.
- If the expression is true, statement S is executed.
- If it is false, the process relinquishes the mutual exclusion and is delayed until B becomes true and no other process is in the region associated with v.
- The critical-region construct guards against certain simple errors associated with the semaphore solution to the critical-section problem that may be made by a programmer.
- Note that it does not necessarily eliminate all synchronization errors; rather, it reduces their number.

Dr R B Kallam

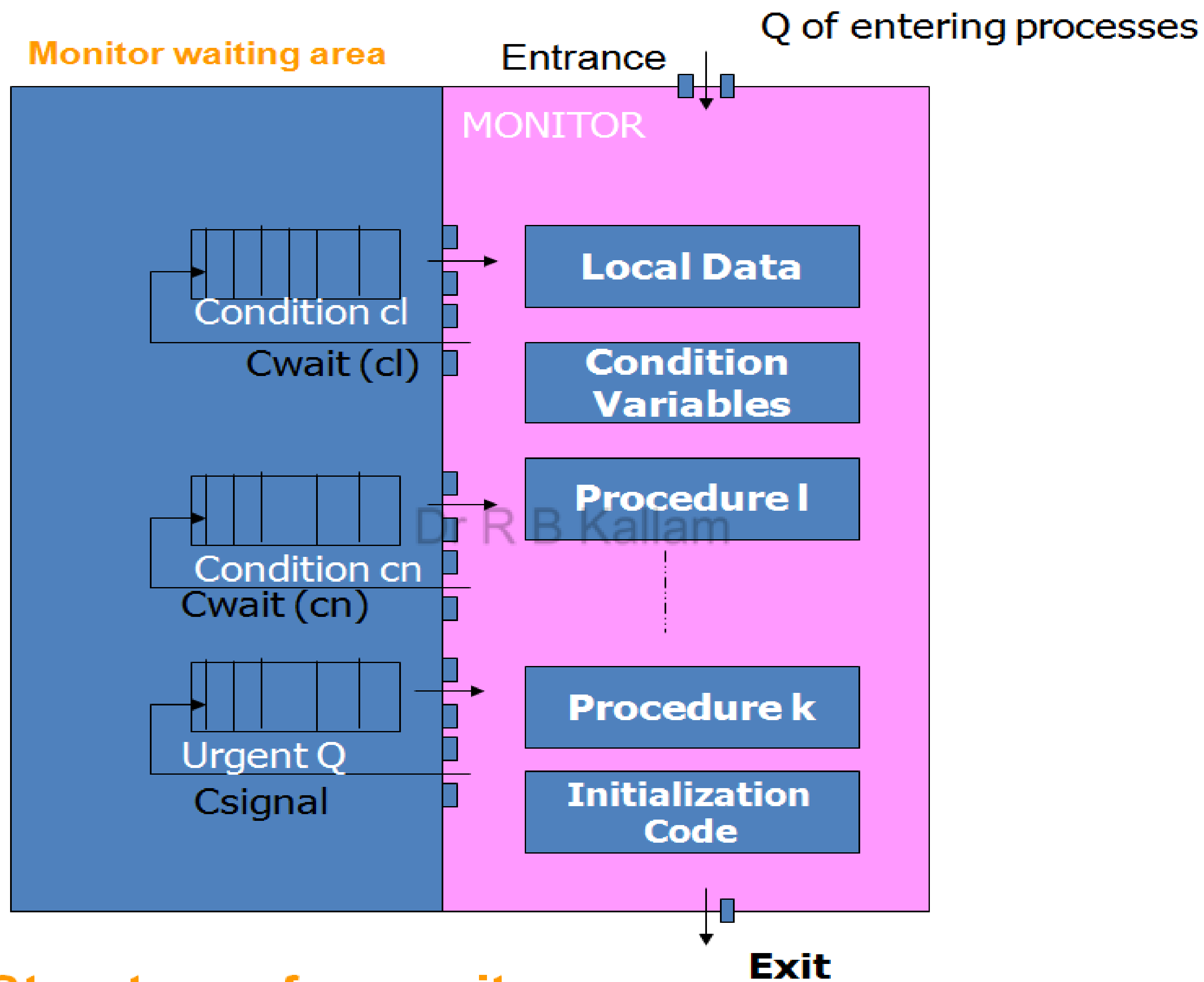
Monitors:

- A monitor is a software module consisting of one or more procedures, an initialization sequence, and local data.
- The chief characteristics of a monitor are the following:
  - The local data variables are accessible only by the monitors procedures and not by any external procedures.
  - A process enters the monitor by invoking one of its procedures.
  - Only one process may be executing in the monitor at a time; any other process that has invoking the monitor is suspended while waiting for the monitor to become available.

Dr.R.B.Kallam

- Monitor is able to provide mutual exclusion.
- The data variables in the monitor can be accessed by only one process at a time.
- Shared data structures / resources can be protected by placing them in a monitor to achieve mutual exclusion.
- A monitor supports synchronization by the use of condition variables that are contained within the monitor and accessible only with in the monitor.
- Although a process can enter the monitor by invoking any of its procedures, we can think of the monitor as having a single entry point that is guarded so that only one process may be in the monitor at a time.
- Other processes that attempt to enter the monitor join a queue of processes blocked waiting for monitor availability.
- Once a process is in the monitor, it may temporarily block itself on condition x by issuing cwait(x) ; it is then placed in a queue of processes waiting to reenter the monitor when the condition changes, and resume execution at the point in its program following the cwait(x) call.
- If a process that is executing in the monitor detects a change in the condition variable x , it issues csignal(x) , which alerts the corresponding condition queue that the condition has changed.

Dr R B Kallam



Structure of a monitor

- **Two functions operate on the condition variables are wait and signal.**

  - Cwait (c): Suspend execution of the calling process on condition c, if another process is already using the same resource of single instance. It means the monitor is now allocated to another process.

  - Csignal (c): Resume execution of some process suspended after a cwait on the same condition. If there are several such processes, choose one of them; if there is no such process do nothing.

Dr R B Kallam

Advantages / operations of Monitors:

- Encapsulate the shared data you wants to protect.
- Acquires the mutex at the start.
- Operates on the shared data.
- Temporarily releases the mutex if it can't complete.
- Reacquires the mutex when it can continue.
- Releases the mutex at the end.

Dr. R B Kallam