CS & IT ENGINEERING

Operating System

REVISION

Process Synchronization (Part-02)

Lecture No.- 06

By- Dr. Khaleel Khan Sir

Recap of Previous Lecture

Topic

Process Synchronization

CS Problem and Requirements

Peterson Solution

Topics to be Covered

Topic Hardware Syncronization

Topic Semaphores

Topic Monitors

Topic

Topic : Peterson's Solution

- Two process solution
- Assume that the load and store machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - int turn;
  - boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section
- The flag array is used to indicate if a process is ready to enter the critical section.
  - flag[i] = true implies that process P_i is ready!

Topic : Algorithm for Process P_i

```
while (true)    {
        flag[i] = true;
        turn = i;
        while (flag[j] && turn = = i);
     << critical section >>
        flag[i] = false;

        /* remainder section */

}
```

If the Two statements are interchanged then M.E is violated;

Topic : Correctness of Peterson's Solution

- Provable that the three CS requirement are met:

1. Mutual exclusion is preserved

   P_i enters CS only if:

   either flag[j] = false or turn = j

2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

Topic : Peterson's Solution and Modern Architecture

- Although useful for demonstrating an algorithm, Peterson's Solution is not guaranteed to work on modern architectures.
  - To improve performance, processors and/or compilers may reorder operations that have no dependencies
- Understanding why it will not work is useful for better understanding race conditions.
- For single-threaded this is ok as the result will always be the same.
- For multithreaded the reordering may produce inconsistent or unexpected results!

Topic : Modern Architecture Example

- Two threads share the data:

  boolean flag = false;

  int x = 0;

- Thread 1 performs

  while (!flag);

  print x

- Thread 2 performs

  x = 100;

  flag = true

- What is the expected output?

If the Two Threads are executed without reordering of Independent Stmnts, then the final value of x is always 100;

If the Statement reordering is possible, then what may be the o/p of x = 0;

Topic : Modern Architecture Example (Cont.)

- However, since the variables flag and x are independent of each other, the instructions:

  ```
  flag = true;
   x = 100;
  ```

  for Thread 2 may be reordered

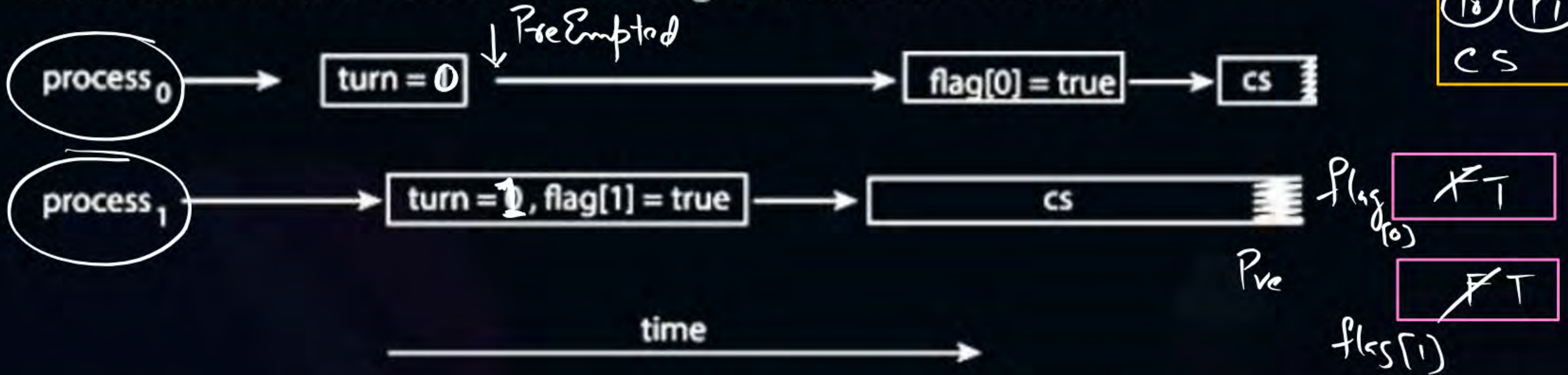
  If this occurs, the output may be 0!

Topic : Peterson's Solution Revisited

turn
$\cancel{0}$ 1

P0 P1
CS

- The effects of instruction reordering in Peterson's Solution



- This allows both processes to be in their critical section at the same time!
- To ensure that Peterson's solution will work correctly on modern computer architecture we must use Memory Barrier.

Topic : Memory Barrier

- Memory model are the memory guarantees a computer architecture makes to application programs.
- Memory models may be either:
  - Strongly ordered – where a memory modification of one processor is immediately visible to all other processors.
  - Weakly ordered – where a memory modification of one processor may not be immediately visible to all other processors.
- A memory barrier is an instruction that forces any change in memory to be propagated (made visible) to all other processors.

Topic : Memory Barrier Instructions

- When a memory barrier instruction is performed, the system ensures that all loads and stores are completed before any subsequent load or store operations are performed.
- Therefore, even if instructions were reordered, the memory barrier ensures that the store operations are completed in memory and visible to other processors before future load or store operations are performed.

Topic : Memory Barrier Example

- Returning to the example of 2 Threads
- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:
- Thread 1 now performs

```
while (!flag)
memory_barrier();
print x
```

- Thread 2 now performs

```
x = 100;
memory_barrier();
flag = true
```

- For Thread 1 we are guaranteed that that the value of flag is loaded before the value of x.
- For Thread 2 we ensure that the assignment to x occurs before the assignment flag.

Topic : Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- We will look at three forms of hardware support:

1. Hardware instructions

2. Atomic variables

TSL/SWAP

Lock

CAS

0

<CS>

Topic : Hardware Instructions

- Special hardware instructions that allow us to either test-and-modify the content of a word, or to swap the contents of two words atomically (uninterruptedly.)
  - Test-and-Set instruction (TSL)
  - Compare-and-Swap instruction

T/F ← TSL (&lock)

Topic : The test_and_set Instruction

- Definition

```
boolean test_and_set (boolean *target)
{
        boolean rv = *target;
        *target = true;
        return rv:
}
```

Current_value of lock

- Properties
  - Executed atomically
  - Returns the original value of passed parameter
  - Set the new value of passed parameter to true

Topic : Solution Using test_and_set()

- Shared boolean variable lock, initialized to false ;
- Solution: Processi

```
do {
     while (test_and_set(&lock)) ; /* do nothing */
     << critical section >>
     lock = false;

     /* remainder section */

} while (true);
```

- Does it solve the critical-section problem?

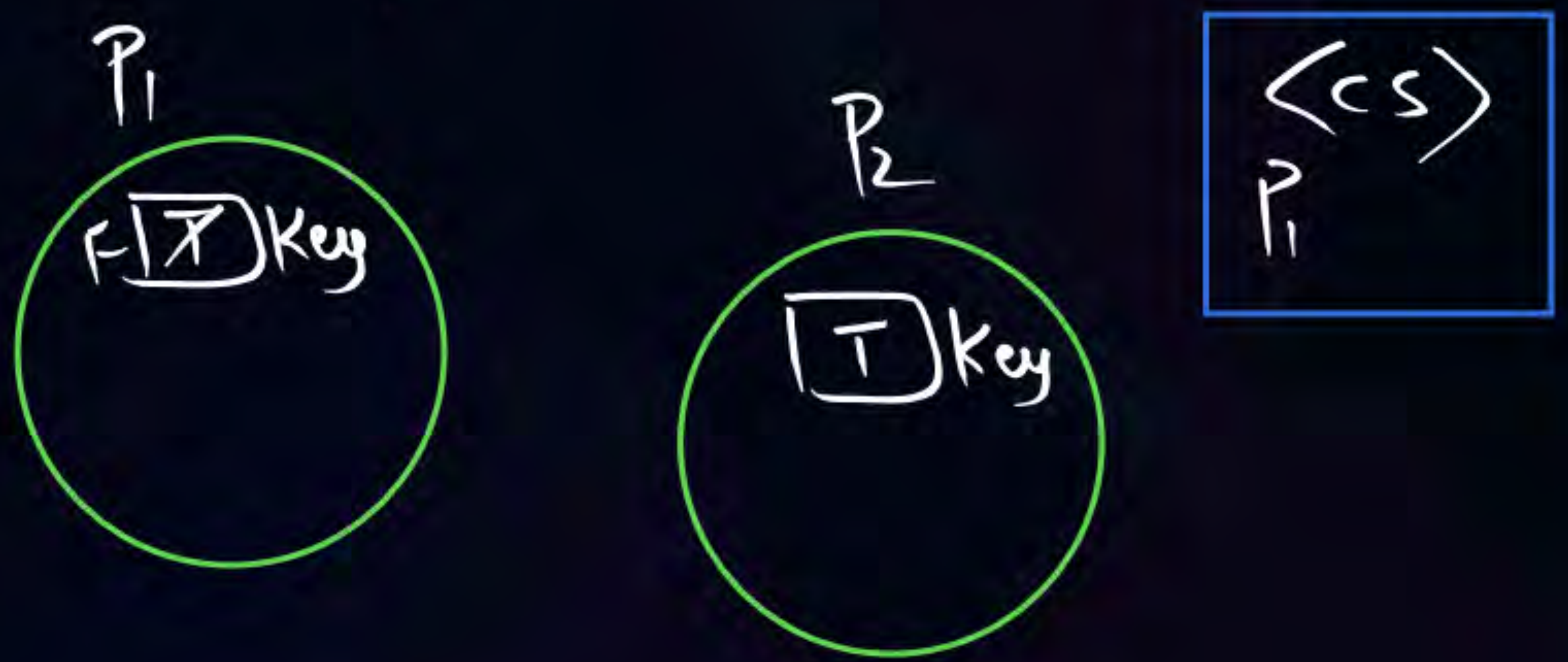
(ATOMIC)

```
void SWAP(Bool *a, Bool *b)
{
    Bool t;

    t = *a;
    *a = *b;
    *b = t;
}
```

lock

$\boxed{\cancel{F}\ T}$




```
void Process(int i)
{
    Bool Key;
    while (1)
    {
        a) Non-CS()
        b) Key = T;
        c) do
           {
               SWAP(&lock, &Key);
           } while(Key==T);
        d) <CS>
        e) lock = F;
    }
}
```

Topic : The compare_and_swap Instruction

(CAS)

- Definition

Lock

```
int compare_and_swap(int *value, int expected, int new_value)
{
    int temp = *value;
    if (*value == expected)
        *value = new_value;
    return temp;
}
```

- Properties
  - Executed atomically
  - Returns the original value of passed parameter value
  - Set the variable value the value of the passed parameter new_value but only if *value == expected is true. That is, the swap takes place only under this condition.

Topic : Solution using compare_and_swap

- Shared integer lock initialized to 0; Pri
- Solution:

```
while (true){
    while (compare_and_swap(&lock, 0, 1) != 0) ; /* do nothing */
    << critical section >>
    lock = 0;
    /* remainder section */
}
```

~~P0~~

P1

1 ~~0~~ lock

P0

<CS>

- Does it solve the critical-section problem?

Topic : Bounded-waiting with compare-and-swap

```
while (true) {
  waiting[i] = true;
  key = 1;
  while (waiting[i] && key == 1)
    key = compare_and_swap(&lock,0,1);
  waiting[i] = false;
  /* critical section */
  j = (i + 1) % n;
  while ((j != i) && !waiting[j])
    j = (j + 1) % n;
  if (j == i)
    lock = 0;
  else
    waiting[j] = false;
  /* remainder section */
}
```

→ TSL

TSL

P_1 P_2 P_3

Topic : Atomic Variables

- Typically, instructions such as compare-and-swap are used as building blocks for other synchronization tools.
- One tool is an atomic variable that provides atomic (uninterruptible) updates on basic data types such as integers and booleans.
- For example:
  - Let sequence be an atomic variable
  - Let increment() be operation on the atomic variable sequence
- The Command:

increment(&sequence);

ensures sequence is incremented without interruption:

(Sequence = Sequence + 1)

Inc

Store

Topic : Atomic Variables

- The increment() function can be implemented as follows:

```
void increment(atomic_int *v)
{
    int temp;
    do {
            temp = *v;
    } while (temp != (compare_and_swap(v,temp,temp+1));
}
```

Topic : Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
  - Boolean variable indicating if lock is available or not
- Protect a critical section by
  - First acquire() a lock
  - Then release() the lock
- Calls to acquire() and release() must be atomic
  - Usually implemented via hardware atomic instructions such as compare-and-swap.
- But this solution requires busy waiting
  - This lock therefore called a spinlock

Topic : Solution to CS Problem Using Mutex Locks

```
while (true) {
        acquire lock
        << critical section >>
        release lock
        << remainder section >>
}
```

Spin-lock

Not available

SEMAPHORES : Dijkstra's Impl

(OS Primitive)

<A.D.T>

- State
  - Int (Value)
    - COUNTING
      - <$-\infty$ to $+\infty$>
    - BINARY
      - 0|1
      - → <CS>
      - (Mutexes)
- → 2 opn's
  - DOWN / wait / P
  - UP / Signal|release / V

(ATOMIC)

Semaphore

Vs

(Condition Variables

Monitor)

(i) Counting Semaphore :

a) DOWN (S) :
- Success : (>= 0)
- Block : (< 0)

b) UP (S) :
→ To wake up a Block process, if any

[<= 0] ; There is atleast one Block Process

CSEM S = 1, T = 1;

P_{r_i}

P(S);
Pre ↔ P(T);

P_{r_j} :

P(T);
P(S);

To solve C.S problem i.e guaranteeing M.E, using Counting Semaphore, the value of Counting Semaphore must be initialized to $\underline{1}$;

Binary Semaphore :

0 1

Topic : Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal(mutex) .... wait(mutex)
  - wait(mutex) ... wait(mutex)
- Omitting of wait (mutex) and/or signal (mutex)
- These – and others – are examples of what can occur when semaphores and other synchronization tools are used incorrectly.

P(S);	P(T);
P(T);	P(S);

Topic : Monitors

- A high-level abstraction that provides a convenient and effective mechanism for 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
- Pseudocode syntax of a monitor:

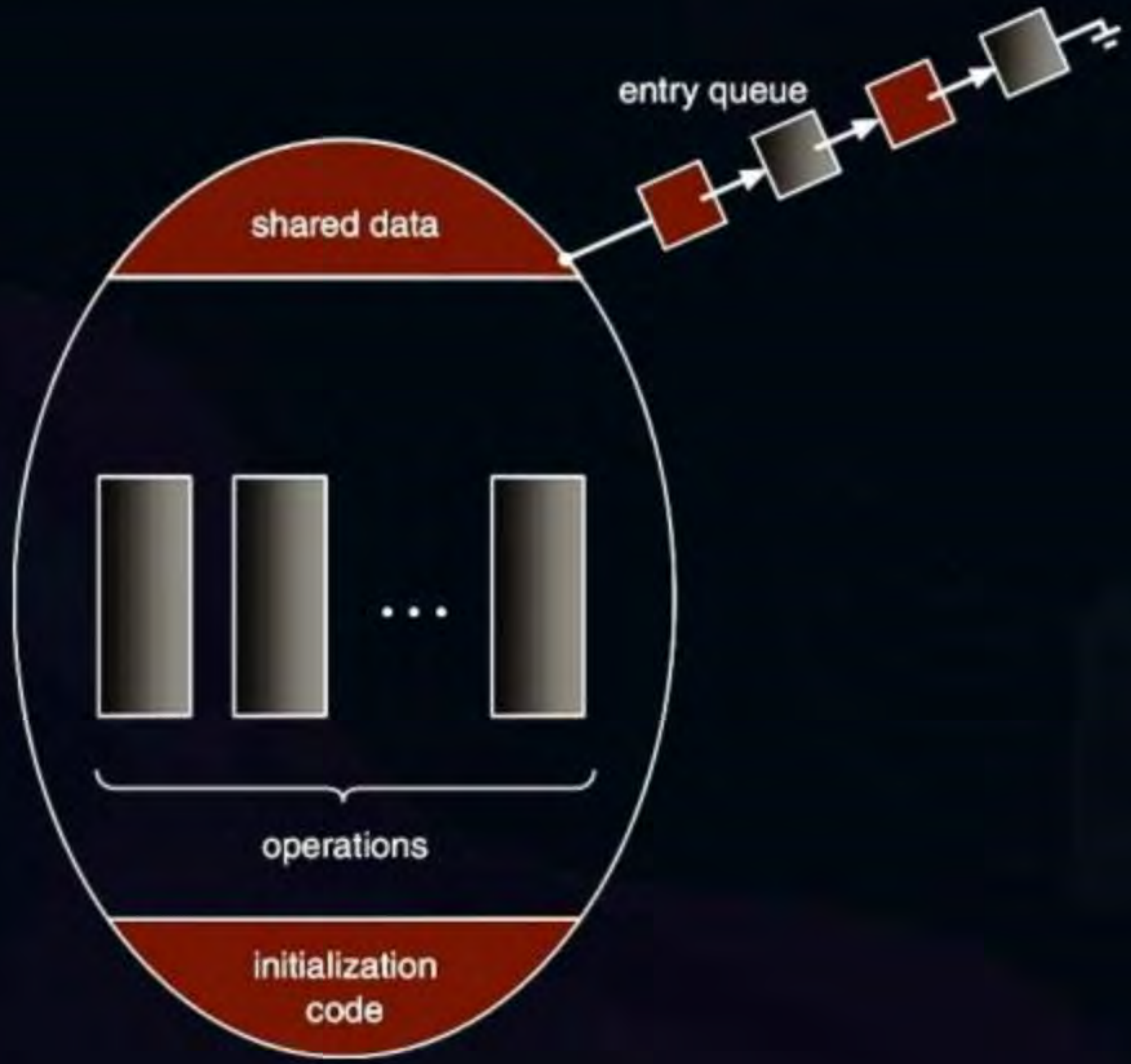
```
monitor monitor-name
{
        // shared variable declarations
        procedure P1 (...) { .... }

        procedure P2 (...) { .... }

        procedure Pn (...) {......}

 initialization code (...) { ... }
}
```

Topic : Schematic view of a Monitor



Topic : Monitor Implementation Using Semaphores

- Variables

  semaphore mutex
  mutex = 1

- Each procedure ***P*** is replaced by

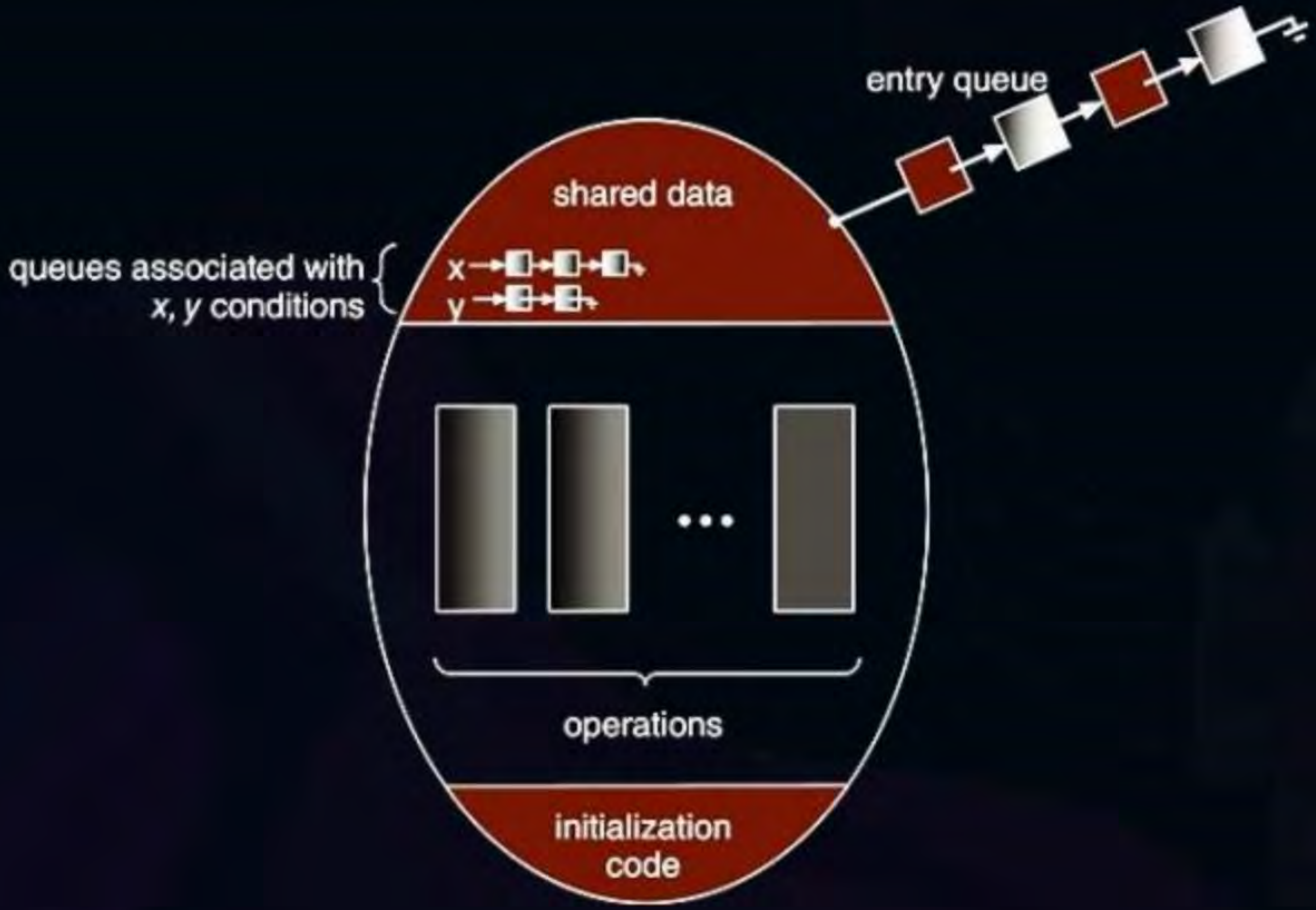
```
wait(mutex);
  ...
body of P;
  ...
signal(mutex);
```

9 Mutual exclusion within a monitor is ensured

Topic : Condition Variables

- condition x, y;
- Two operations are allowed on a condition variable:
  - x.wait() – a process that invokes the operation is suspended until x.signal()
  - x.signal() – resumes one of processes (if any) that invoked x.wait()
  - If no x.wait() on the variable, then it has no effect on the variable

Topic : Monitor with Condition Variables



Topic : Usage of Condition Variable Example

- Consider P_1 and P_2 that that need to execute two statements S_1 and S_2 and the requirement that S_1 to happen before S_2
  - Create a monitor with two procedures F_1 and F_2 that are invoked by P_1 and P_2 respectively
  - One condition variable "x" initialized to 0
  - One Boolean variable "done"
  - **F1:**

```
S1;
done = true;
x.signal();
```

  - **F2:**

```
if done = false
x.wait()
S2;
```

Topic : Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next_count = 0; // number of processes waiting inside the monitor
```

- Each function P will be replaced by

```
wait(mutex);
   ...
   body of P;
   ...
if (next_count > 0)
   signal(next)
else
   signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Topic : Implementation – Condition Variables

- For each condition variable *x*, we have:

```
semaphore x_sem; // (initially  = 0)
int x_count = 0;
```

- The operation x.wait() can be implemented as:

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

Topic : Implementation (Cont.)

- The operation x.signal() can be implemented as:

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

Topic : Resuming Processes within a Monitor

- If several processes queued on condition variable **x**, and **x.signal()** is executed, which process should be resumed?
- FCFS frequently not adequate
- Use the **conditional-wait** construct of the form **x.wait(c)**

where:

- **c** is an integer (called the priority number)
- The process with lowest number (highest priority) is scheduled next

Topic : Single Resource allocation

- Allocate a single resource among competing processes using priority numbers that specifies the maximum time a process plans to use the resource

```
R.acquire(t);
        ...
        access the resurce;
        ...

R.release;
```

- Where R is an instance of type Resource Allocator

Topic : Single Resource allocation

- Allocate a single resource among competing processes using priority numbers that specifies the maximum time a process plans to use the resource
- The process with the shortest time is allocated the resource first
- Let R is an instance of type ResourceAllocator (next slide)
- Access to ResourceAllocator is done via:

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

- Where t is the maximum time a process plans to use the resource

Topic : A Monitor to Allocate Single Resource

```
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;
        }
}
```

Topic : Single Resource Monitor (Cont.)

- Usage:

  acquire

  ...

  release

- Incorrect use of monitor operations
  - release() ... acquire()
  - acquire() ... acquire())
  - Omitting of acquire() and/or release()

Topic : Liveness

- Processes may have to wait indefinitely while trying to acquire a synchronization tool such as a mutex lock or semaphore.

Starvation

- Waiting indefinitely violates the progress and bounded-waiting criteria ~~discussed at the beginning of this chapter~~.

Synch. Mech

- LIVENESS PROPERTY : refers to a set of properties that a system must satisfy to ensure processes make progress.
- Indefinite waiting is an example of a **LIVENESS FAILURE**.

Topic : Liveness

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

```
P0                  P1
wait(S);            wait(Q);
wait(Q);            wait(S);
 ...                 ...
signal(S);          signal(Q);
signal(Q);          signal(S);
```

B   B

Liveness Failure

- Consider if P_0 executes wait(S) and P_1 wait(Q). When P_0 executes wait(Q), it must wait until P_1 executes signal(Q)
- However, P_1 is waiting until P0 execute signal(S).
- Since these signal() operations will never be executed, P_0 and P_1 are deadlocked.

Topic : Liveness

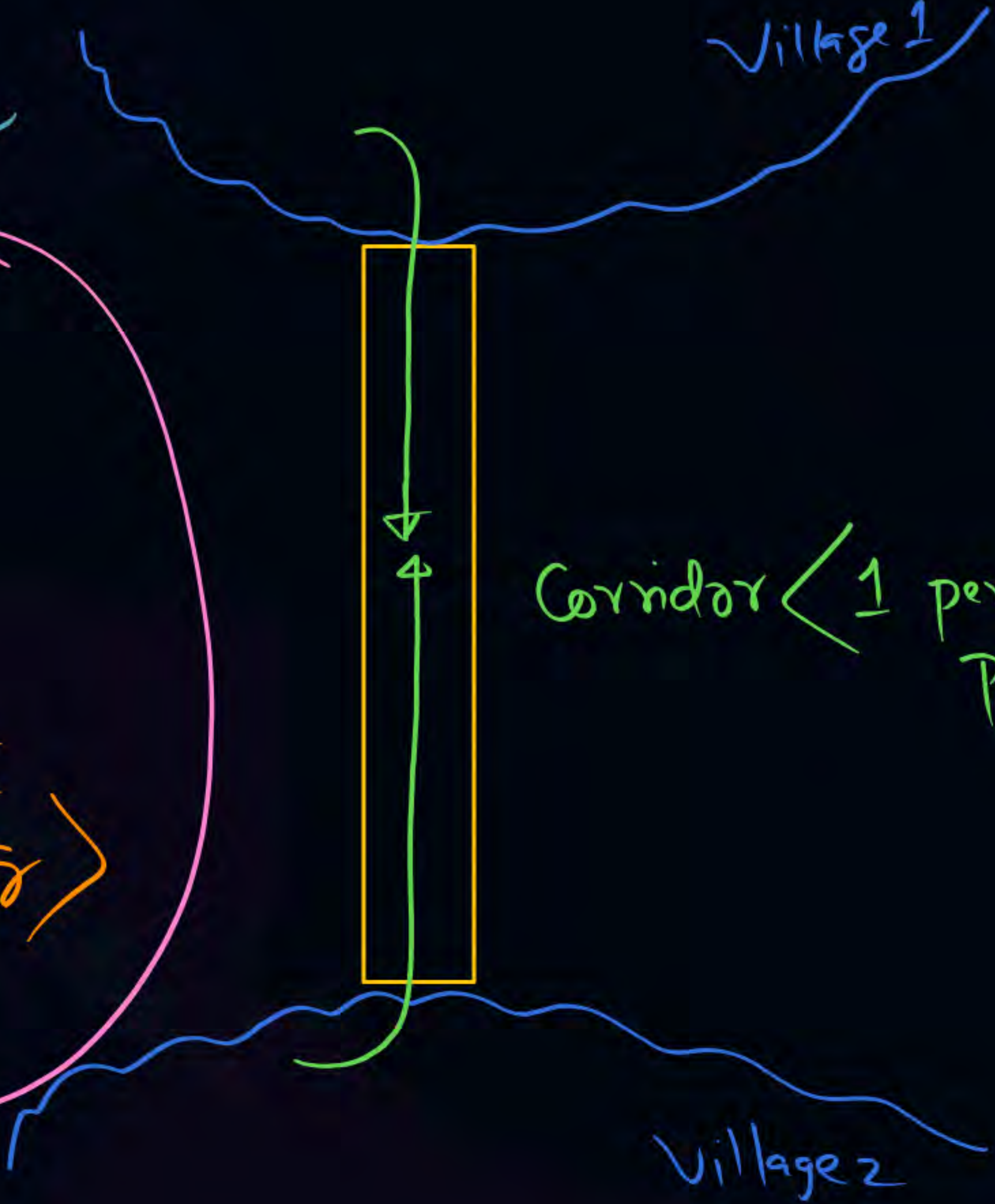
- Other forms of deadlock:
- Starvation – indefinite blocking
  - A process may never be removed from the semaphore queue in which it is suspended   ⟨ Q → LIFO ⟩
- Priority Inversion – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via priority-inheritance protocol

L   H

CS $\boxed{L}$

Liveness Failure

livelock,
deadlock,
Starvation
<Indefinite waiting>



Corridor <1 person could pass thru it>

Conceptually livelock & deadlock are same

2 mins Summary

Topic	One	Pet. Soln & Mod. Arch, Mem. Barriers
Topic	Two	H/w Synch
Topic	Three	TSL
Topic	Four	SWAP, CAS,
Topic	Five	SEMAPHORES, Liveness Property

THANK - YOU