UNIT : 6
MUTUAL EXCLUSION

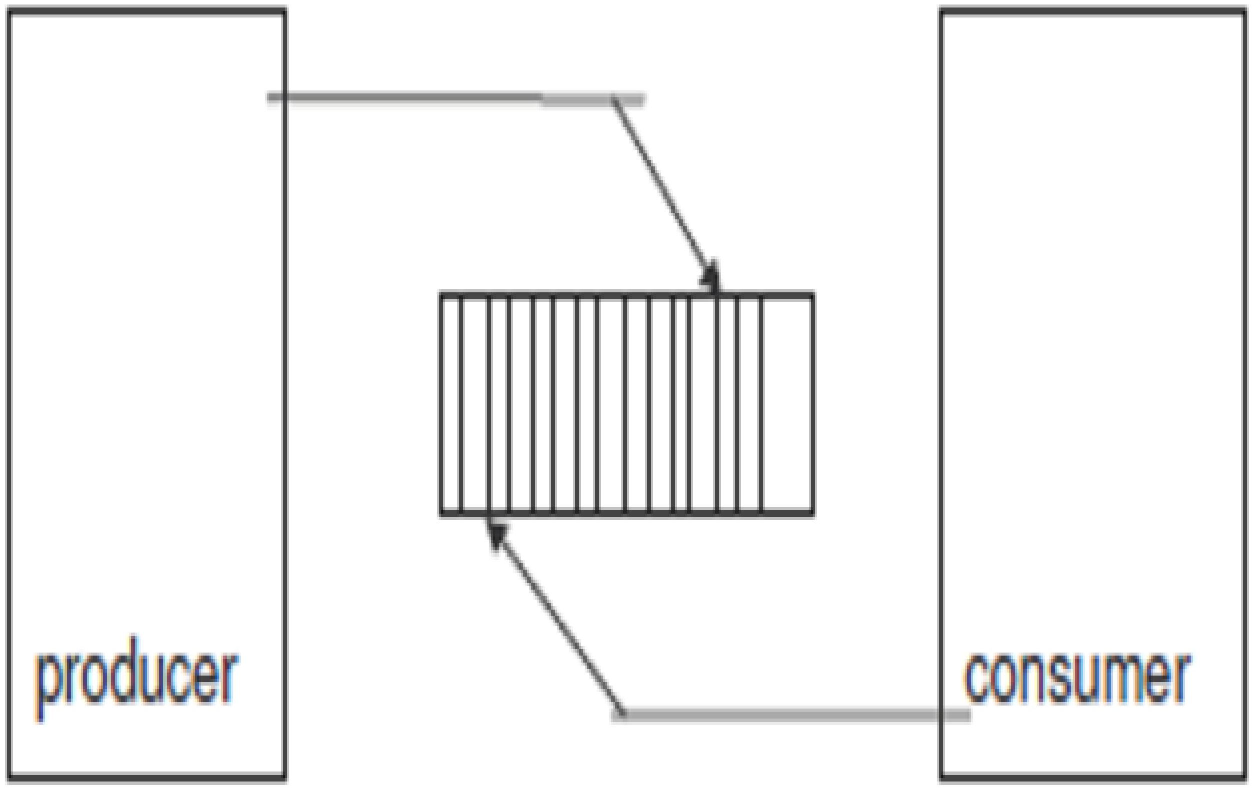
Mutual Exclusion Revisited : Critical Sections

- Concurrent access to shared data may result in data inconsistency (e.g., due to race conditions)

- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

- 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 **count** that keeps track of the number of full buffers.
  - Initially, count is set to 0.
  - Incremented by producer after producing a new buffer
  - Decremented by consumer after consuming a buffer

Mutual Exclusion Revisited : Critical Sections



Mutual Exclusion Revisited : Critical Sections

- Mutual exclusion is required for shared memory.
- Mutual exclusion must be ensured whenever there is a shared area of memory and processes writing to it.
- The main motivation is to avoid race condition among processes.

Mutual Exclusion Revisited : Critical Section – Race Condition

- count++ could be implemented as

```
register1 := count
register1 := register1 + 1
count     := register1
```

- count-- could be implemented as

```
register2  := count
register2  := register2 - 1
count      := register2
```

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

```
T0: producer execute  register1 := count
{register1 = 5}
T1: producer execute  register1 := register1 + 1
{register1 = 6}
T2: consumer execute  register2 := count
{register2 = 5}
T3: consumer execute  register2 := register2 - 1
{register2 = 4}
T4: producer execute      count := register1
{count = 6}
T5: consumer execute      count := register2
{count = 4}
```

Mutual Exclusion Revisited : Critical Sections

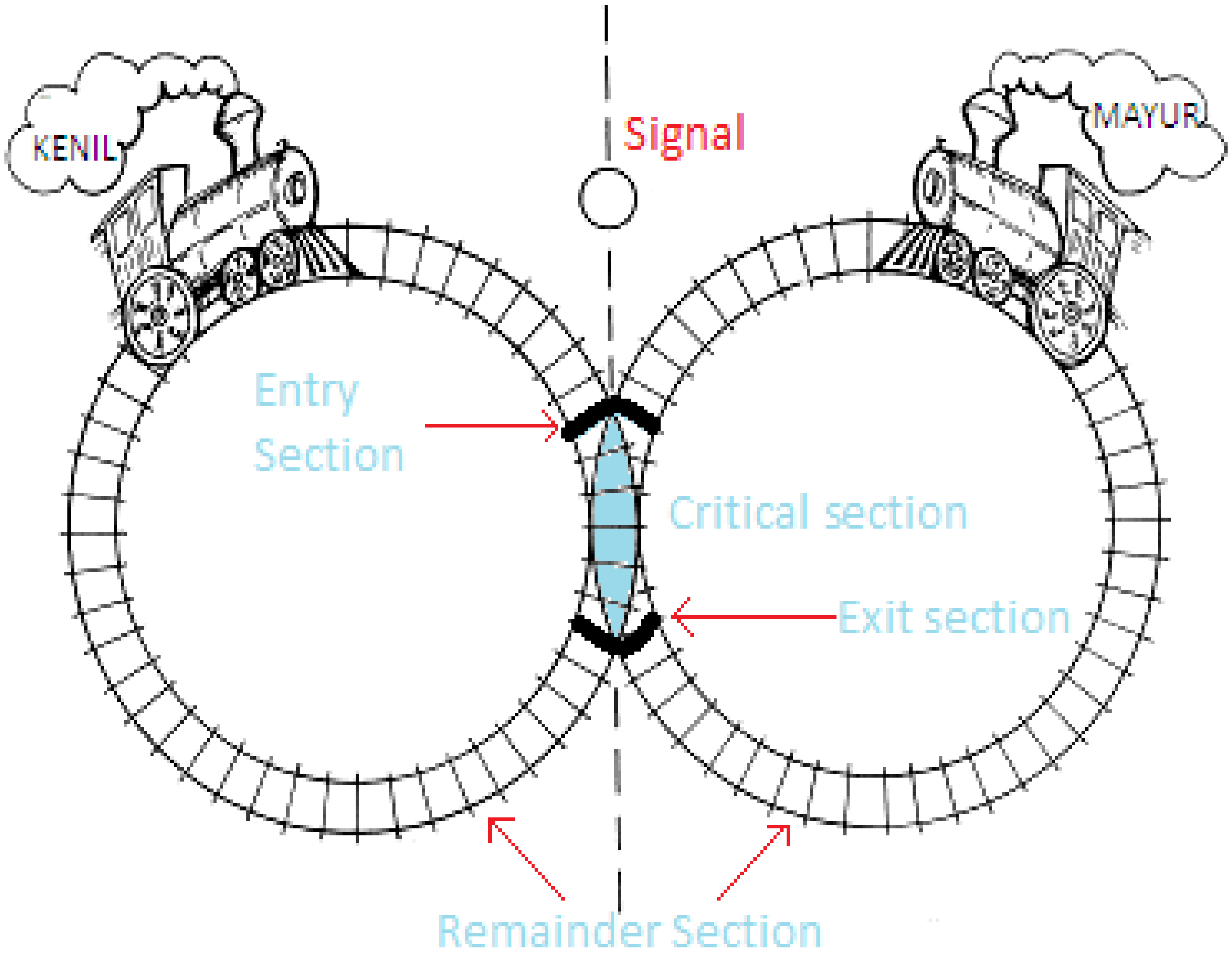
- Critical Section is the section of code that is executed exclusively and without any interruptions – none of its operations can be ignored.

- Unix provides a facility called semaphore to allow processes to use critical sections mutually exclusive of each other.

Mutual Exclusion Revisited : Critical Sections

- A section of code, common to n cooperating processes, in which the processes may be accessing common variables.
- A Critical Section Environment contains:
  - Entry Section   Code requesting entry into the critical section.
  - Critical Section   Code in which only one process can execute at any one time.
  - Exit Section   The end of the critical section, releasing or allowing others in.
  - Remainder Section   Rest of the code AFTER the critical section.

Mutual Exclusion Revisited : Critical Sections



Solutions to critical section problem

Mutual Exclusion - No two processes eat simultaneously

Progress - If no process eats forever, and some process is hungry, then some (potentially different) hungry process eventually eats.

Bounded Waiting- A bound exists on the number of times that other processes are allowed to eat after a process P becomes hungry and before process P eats.

- Assume that each process executes at a nonzero speed
- No assumption concerning relative speed of the N processes

SEMAPHORES

- A semaphore is essentially a variable which is treated in a special way.
- Access and operations on a semaphore is permitted only when it is in a free state.
- If a process locks a semaphore, others cannot get access to it.

SEMAPHORES

- When a process enters a critical section, other processes are prevented from accessing this shared variable.
- A process frees the semaphore on exiting the critical section.
- To ensure this working, a notion of atomicity or indivisibility is invoked.

SEMAPHORES

- Synchronization tool that does not require busy waiting
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()
  - Originally called P() and V()
- Less complicated
- Can only be accessed via two indivisible (atomic) operations
  - wait (S) {

    ```
    wait (S) {
          while S <= 0
               ; // no-op
             S--;
     }
    ```
  - signal (S) {

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

Semaphores

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion
  - Semaphore S; // initialized to 1
  - wait (S);

            Critical Section

        signal (S);

Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time.
- Thus, 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.

Basic Properties of Semaphores

- A semaphore takes only integer values.

- There are only two operations possible on a semaphore :-

  - A wait operation on a semaphore decreases its value by 1.
  wait(s) : while s < 0 do loop; s := s - 1;

  - A signal operation increments its value
  signal(s) : s := s + 1;

Basic Properties of Semaphores

- A semaphore operation is atomic.

  A process is blocked if its wait operation evaluates a negative semaphore value.

- A blocked process can be unblocked when some

  other process executes a signal operation.

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.

SEMAPHORE IMPLEMENTATION WITH NO BUSY WAITING

- Implementation of wait:

```
wait (S){
    value--;
    if (value < 0) {
        add this process to waiting queue
        block(); }
}
```

- Implementation of signal:

```
Signal (S){
    value++;
    if (value <= 0) {
        remove a process P from the waiting queue
        wakeup(P); }
}
```

Deadlock and starvation

- 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

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Usage of Semaphore

- Suppose two processes P1 and P2 use a semaphore variable with initial value 0.

- We assume both processes have a program structure as:

```
repeat
        some process code here
        wait(use);
enter the critical section the process
        (manipulates a shared area);
signal(use);
rest of the process code;
until false;
```

USAGE OF SEMAPHORE

- We have here an infinite loop for both processes.

- Either P1 or P2 can be in its critical section.

Usage of semaphore

- The following is a representative operational sequence.
- Initially neither process is in critical section and use = 0.
- P1 arrives at critical section first and calls wait(use).
- It succeeds and enters the critical section setting use = -1.
- P2 wants to enter its critical section. Calls wait procedure.
- As use < 0, P2 busy waits.
- P1 executes signal and exits its critical section, use = 0 now.
- P2 exits busy wait loop. It enters critical section use = -1.
- The above sequence continues.

Usage of semaphore

- Semaphore is also used to synchronize amongst processes. A process may have a synchronizing event.

USAGE OF SEMAPHORE

- Suppose we have 2 processes Pi and Pj, Pj can execute some statement sj only after statement si in Pi has been executed.

- This can be achieved with semaphore se initialized to -1 as follows:
    - In Pi, execute sequence sj ; signal(se);
    - In Pj execute wait(se); sj;

- Now, Pj must wait completion of sj before it can execute sj.

Usage of semaphore

- These resources are not all used all the time.
- In case of a printer - output resource is used once in a while.
- This printer must be used amongst multiple users – because the printer is expensive and because it is sparingly used.

Usage of semaphore

- Resources may be categorized depending upon the nature of their use.
- OS needs a policy to schedule its use - dependant on nature of use, frequency and context of use.
- For a printer, OS can spool the data as the printer requests.

USAGE OF SEMAPHORE

- Each printer job must have exclusive use of it till it finishes.
- Print-outs would be garbled otherwise.
- Some times processes may require more than one resource.
- A process may not be able to proceed till it gets all the resource.

USAGE OF SEMAPHORE

- Consider a process P1 requiring resources r1 and r2. Consider process P2 requiring resources r2 and r3. P1 will proceed only when it has both r1 and r2. P2 needs both r2 and r3. If P2 has r2, then P1 has to wait until P2 releases r2 or terminates.