# Inter-Process Communication and Synchronization of Processes, Threads and Tasks:

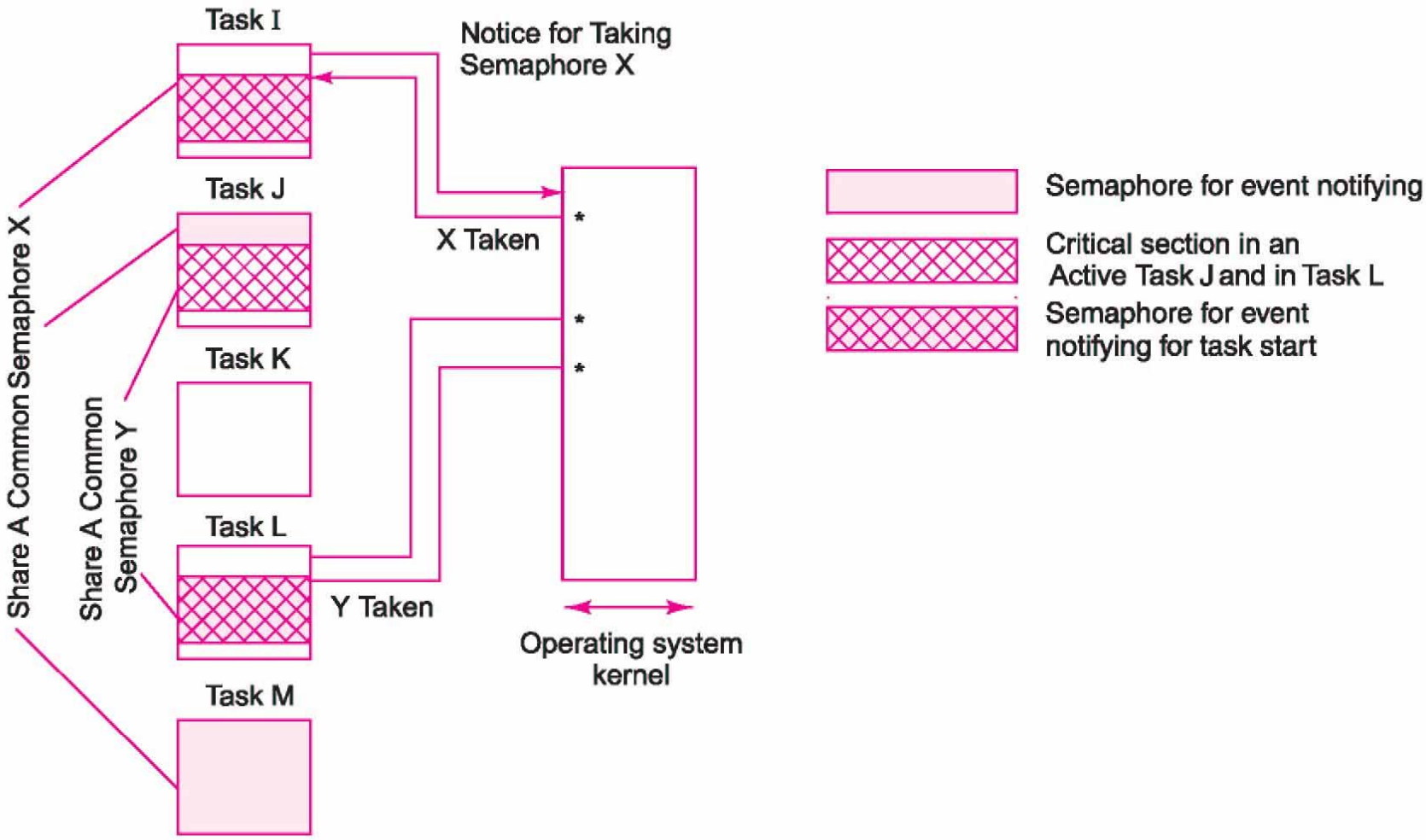
# Lesson-8: Use of Multiple Semaphores and counting Semaphore for Synchronizing the Tasks

# Use of Multiple Semaphores

# Use of Multiple Semaphores for Synchronizing the Tasks

- Example of the use of two semaphores for synchronizing the tasks I, J and M and the tasks J and L, respectively

# Use of two semaphores for synchronizing tasks I, J, K, L and M




Task K Blocks when Y is taken by L
Task J and M Block when X is taken by L

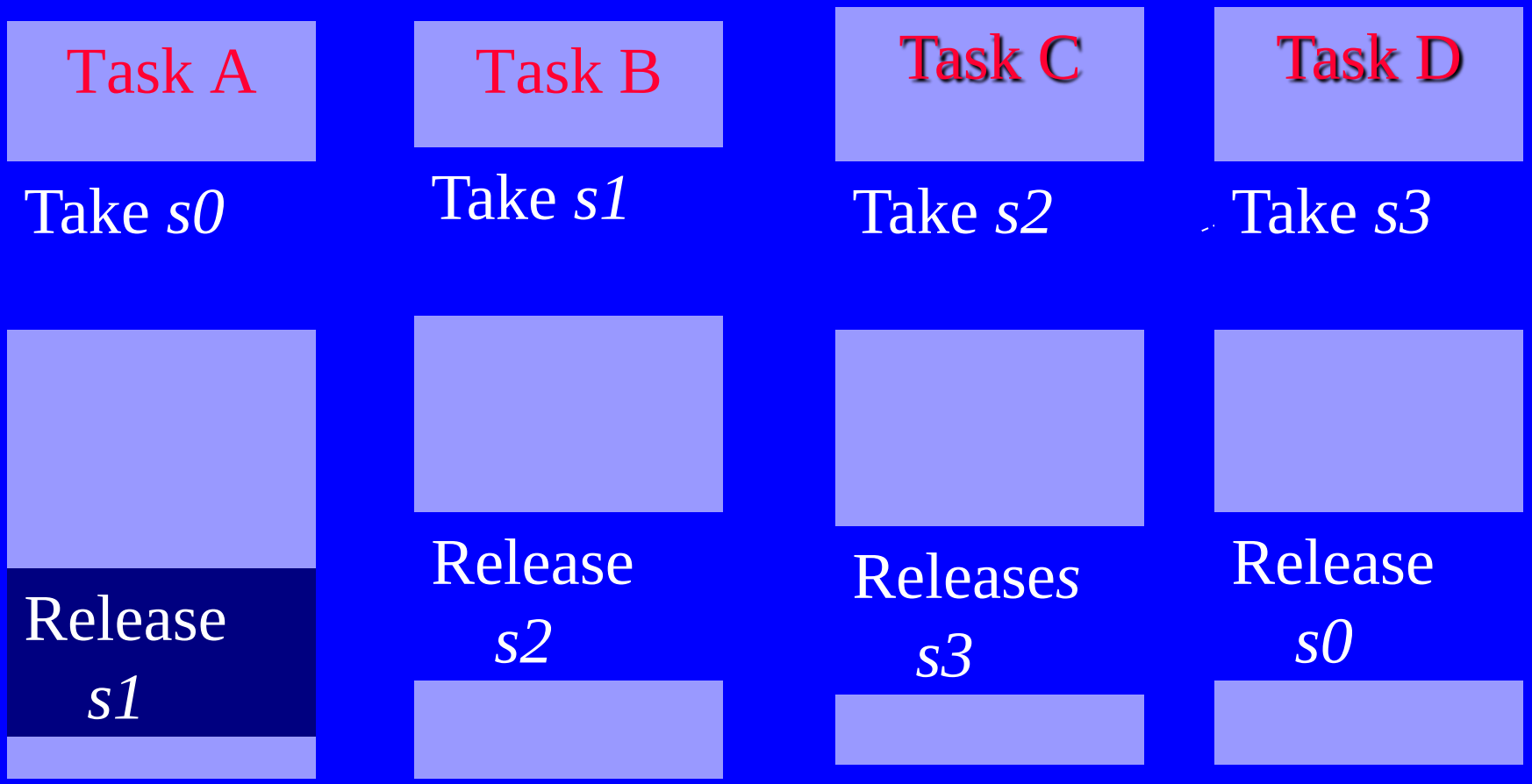
# OS Functions for Semaphore

- OSSemPost ( )─ an OS IPC function for posting a semaphore and assume OSSemPend ( ) ─ another OS IPC function for waiting for the semaphore.
- Let *sTask* is the mutex semaphore pending and posted at each task to let another run.
- Let *sTask1* initially is 1 and *sTask2*, *sTask3* and *sTask4* initially are 0s

# Codes

- Consider Codes such that firstly task *I* will run, then *J*, then *K*, then *L*, then *I* when at an initial instance *sTask1* = 1 and *sTask2* = *sTask3* = *sTask4* = 0

# Running of Tasks A, B, C, and D Synchronized through IPCs s0, s1, s2 and s3



# Running of Tasks A, B, C, and D Synchronized through IPCs s0, s1, s2 and s3

• Task A sends an IPC s1, B is waiting for s1, when s1 releases, B takes s1 and runs. Similarly, C runs on taking s2, D runs on taking s3, again A runs on taking s0. Running of the codes of tasks A to D synchronizes using the IPCs

# Codes for task *I* wait for running

```
static void Task_ I  (void *taskPointer)  {
.
while (1) {
OSSemPend (sTask1) /* Post the
   semaphore sTask1. Means that OS
   function decrements sTask1 in
   corresponding event control block.
   sTask1  becomes 0 and following code
   run*/
```

# Codes for task *I* run and release semaphore for *J*

```
/* Codes for Task_I */
.OSSemPost (sTask2) /* Post the
   semaphore sTask2. This means that OS
   function increments sTask2 in
   corresponding event control block.
   sTask2 becomes 1 */
};
```

# Codes for task *J* wait for semaphore from *I*

- static void Task_ J (void *taskPointer) **{**

.

while (1) {

OSSemPend (sTask2) /* Wait sTask2. Means wait till sTask2 is posted and becomes 1. When sTask2 becomes 1 and the OS function decrements sTask2 in corresponding event control block, sTask2 becomes 0. Task then runs further the following code*/

# Codes for task J run and release semaphore for K

- /* Code for Task J */

.

.

OSSemPost (sTask3) /* Post the semaphore sTask3. Means that OS function increments sTask3 in corresponding event control block. sTask3 becomes 1. */

};

# Codes for task K wait for semaphore from J

- static void Task_ K (void *taskPointer) {

.

while (1) {

OSSemPend (sTask3) /* Wait for the semaphore sTask3. Means that wait till sTask3 is posted and becomes 1. When sTask3 becomes 1 and the OSSemPend decrements sTask3 in corresponding event control block. sTask3 becomes 0. Task then runs further the following code*/

# Codes for task K run and release semaphore for L

/* Code for Task K */

.

.

OSSemPost (sTask4) /* Post the semaphore sTask4. This means that OS function increments sTask4 in corresponding event control block. sTask4 becomes 1. */

};

# Codes for task L wait for semaphore from K

```
static void Task_ L  (void *taskPointer)  {
.
while (1) {
OSSemPend (sTask4) /* Wait for the semaphore
    sTask4. This means that task waits till sTask4 is
    posted and becomes 1. When sTask4 becomes 1
    and the OS function is to decrements sTask3 in
    corresponding event control block. sTask4
    becomes 0. Task then runs further the following
    code*/
```

# Codes for task L run and release semaphore for I

- /* Code for Task L */
- .
- .
- OSSemPost (sTask1) /* Post the semaphore sTask1. This means that OS function increments sTask1 in corresponding event control block. sTask1 becomes 1. */
- };

# Number of tasks waiting for same semaphore

# Number of tasks waiting for Same Semaphore

- OS Provides the answer
- In certain OS, a semaphore is given to the task of highest priority among the waiting tasks.
- In certain OS, a semaphore is given to the longest waiting task (FIFO mode).

# Number of tasks waiting for Same Semaphore

- In certain OS, a semaphore is given as per selected option and the option is provided to choose among priority and FIFO.
- The task having priority, if started takes a semaphore first in case the priority option is selected. The task pending since longer period takes a semaphore first in case the FIFO option is selected.

# Counting Semaphore

# OS counting semaphore functions

- Counting semaphore *scnt* is an unsigned 8 or 16 or 32 bit-integer.
- A value of *scnt* controls the blocking or running of the codes of a task.
- *scnt* decrements each time it is taken.
- *scnt* increments when released by a task.

# Counting-semaphore

- *scnt* at an instance reflects the initialized value minus the number of times it is taken plus the number of times released.
- *scnt* can be considered as the number of tokens present and the waiting task will do the action if at least one token is present.
- The use of *scnt* is such that one of the task thus waits to execute the codes or waits for a resource till at least one token is found

# Counting Semaphore application example

- Assume that a task can send on a network the stacks into 8 buffers.
- Each time the task runs it takes the semaphore and sends the stack in one of the buffers, which is next to the earlier one.
- Assume that a counting semaphore *scnt* is initialized = 8. After sending the data into the stack, the task takes the *scnt* and *scnt* decrements. When a task tries to take the *scnt* when it is 0, then the task blocks and cannot send into the buffer

# ACVM Example

- Consider *Chocolate delivery task*.
- It cannot deliver more than the total number of chocolates, *total* loaded into the machine.
- Assume that a semCnt is initialized equal to *total*.
- Each time, the new chocolates loaded in the machine, semCnt increments by the number of new chocolates.

# *Chocolate delivery task* code

```
static void Task_Deliver  (void *taskPointer)  {
.
while (1) {       /* Start an infinite while-loop. */
/* Wait for an event indicated by an IPC from Task
   Read-Amount */
.
If (Chocolate_delivered) OSSemPend (semCnt) /* If
   chocolate delivered is true, if semCnt is not 1 or >
   1 (which means is 0 or less) else decrement the
   semCnt and continue remaining operations */
.
};
```

# Summary

# We learnt

- Multiple semaphores can be used in multitasking system
- Different set of semaphores can share among different set of tasks.
- Semaphore provides a mechanism to synchronize the running of tasks

# We learnt

- Counting semaphore provides a way of taking it and releasing it number of times
- When taken by a waiting task section when it is 1 or > 1, it decrements, the semaphore becomes available
- It increments when posts (sent or released) a task .

# We learnt

- Counting semaphore is an unsigned integer semaphore that can be ‘taken’ till its value = 0 and is usually initialized to a high value.
- It can also be ‘given’ (sent or posted) a number of times.

End of Lesson 8 of Chapter 9 on

Use of Multiple Semaphores and counting Semaphore for Synchronizing the Tasks