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

Lesson-7: Concept of Semaphore as resource key

Semaphore as a resource key and for critical sections having shared resource (s)

Shared Resource (s)

- Shared memory buffer is to be used only by one task (process or thread) at an instance
- Print buffer, global variable (s), file, network, LCD display line or segment, …are also used only by one task (process or thread) at an instance

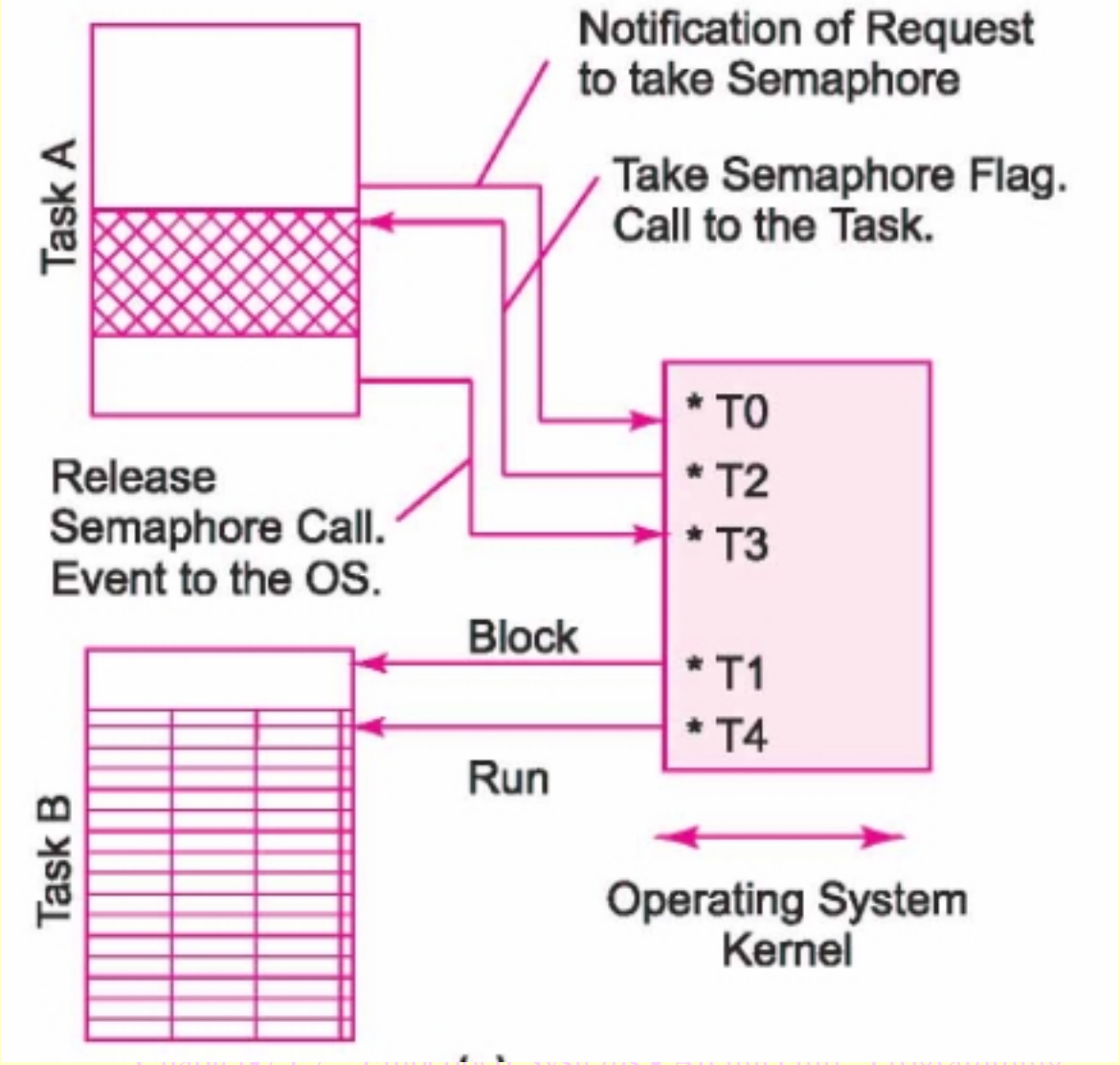
OS Functions for Semaphore as a resource key

- OS Functions provide for the use of a semaphore resource key for running of the codes in critical section
- Let a binary Boolean variable, *sm*, represents the semaphore.
- Resource key can be for shared memory buffer, print buffer, global variable (s), file, network, LCD display line or segment, …which is to be used only by one task (process or thread) at an instance

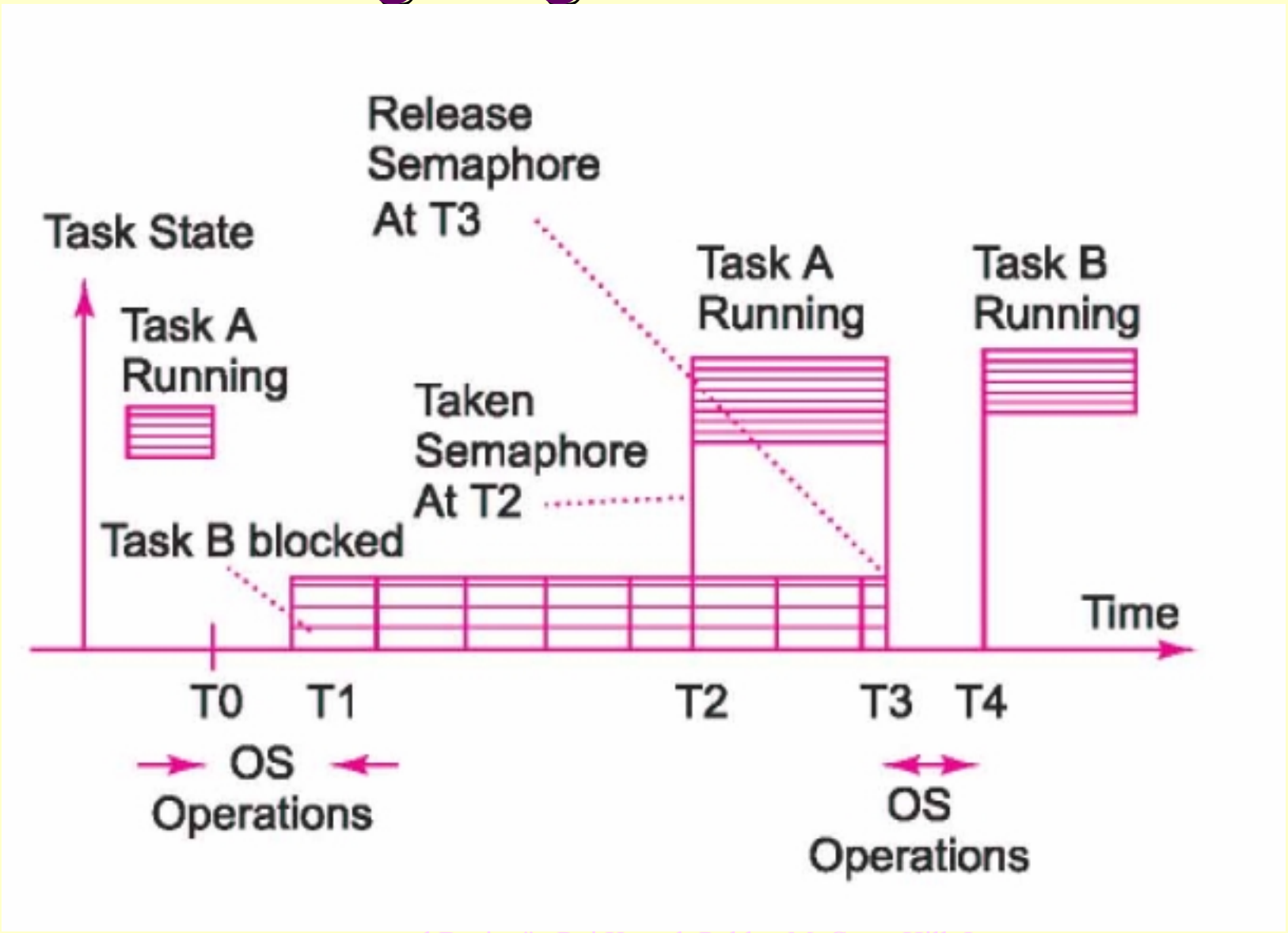
OS Functions for Semaphore as a resource key

- The taken and post operations on *sm*— *(i)* signals or notifies operations for starting the task section using a shared resource (ii) signals or notifies operations for leaving the task section after using a shared resource .
- Semaphore function's variable *sm* is like a key for the resource — (i) beginning of using the shared resource is by taking the key (ii) Release of key is the end of using the shared resource

Use of semaphore between task A and B



Timing Diagram for the states



and Design" , Raj Kamal, Publs.: McGraw-Hill, Inc.

Critical Section

- Section of codes using for shared memory buffer, print buffer, global variable (s), file, network, LCD display line or segment, …which is to be used only by one task (process or thread) at an instance, which can also be used by another section also at another instance

Mutex Semaphore for use as resource key

Mutex

- Mutex means mutually exclusive key
- Mutex is a binary semaphore usable for protecting use of resource by other task section at an instance
- Let the key *sm* initial value = 1

Mutex

- When the key is taken by section the key *sm* decrements from 1 to 0 and the waiting task codes starts.
- Assume that the *sm* increments from 0 to 1 for signaling or notifying end of usie of the key that section of codes in the task or thread.

Mutex Semaphore…

- When sm = 0 — assumed that it has been taken (or accepted) and other task code section has not taken it yet and using the resource
- When sm = 1— assumed that it has been released (or sent or posted) and other task code section can now take the key and use the resource
-

Mutex Semaphore use in ISR and Task

- An ISR does not take the key
- A task can wait for taking the key and release the key

Example of UpDate_Time and Read_Time Tasks in ACVM

Uses in UpDate_Time and Read_Time Tasks in ACVM

- An interrupt service routine runs when timer timeouts and makes posts key initial value = 1 at time T1
- *Update_Time task* — When the task updates t information at the time device, it has to notify to the *Read_Time task* not to run a waiting section of the code to read t from the time device as it is being updated

Uses in UpDate_Time and Read_Time Tasks in ACVM

- *Read_Time task* — runs a waiting section of the code to read t from the time device after updating of the time and date

Uses in key (mutex) …

- Assume OSSemPend ( ) ─ another OS IPC function for waiting for the semaphore.
- OSSemPost ( )─ is an OS IPC function for posting a semaphore

Uses as key (mutex)…

- Let *supdateT* is the binary semaphore posted from *Chocolate delivery task* and taken by a *Display task* section for displaying thank you message.
- Let *supdateT* initial value = 0.

Wait for the update key after the ISR posts time …

static void Task_ *Update_Time (void *taskPointer)* **{**

.

while (1) {

.

OSSemPend (*supdateT*) /* Wait the semaphore *supdateT =1 posted by* I_S. This means that OS function decrements *sdispT* in corresponding event control block. *supdateT* becomes 0 at T2. */

Post of the update key after updating time and date …

/* Codes for writing into the time device. */

.

OSSemPost (*supdateT*) /* Post the semaphore *supdateT*. This means that OS function increments *supdateT* in corresponding event control block. *supdateT* key becomes 1 at instance T3. */

.

};

Key Taking after Update task posting the key

static void Task_*Read_Time (void *taskPointer)* **{**

.

while (1) {

.

OSSemPend (*sdispT*) /* Wait *supdateT*. Means *supdateT* is posted by the update task and becomes 1. When *supdateT* becomes 1 and the OS function decrements *supdateT and* becomes 0 at instance T4. Task then runs further the following code*/

Key Release after reading time and date

- /* Code for reading the time device */
- .
- .
- OSSemPost (*supdateT*) /* Post the semaphore *supdateT*. This means that OS function increments *supdateT* in corresponding event control block. *supdateT* becomes 1 at instance T5. */
- *};*

Summary

We learnt

- Semaphore provides a mechanism to let a section of the task code use a resource after key is available.

We learnt

- Provides a way of using key for start and end of the code.
- Semaphore decrements when accepted or taken by waiting task section after the key becomes available and increments when posts (sent or released) that key .

End of Lesson 7 of Chapter 7