# Real-time Systems

# Chapter 3: Task and Task Synchronization

Ngo Lam Trung

Dept. of Computer Engineering

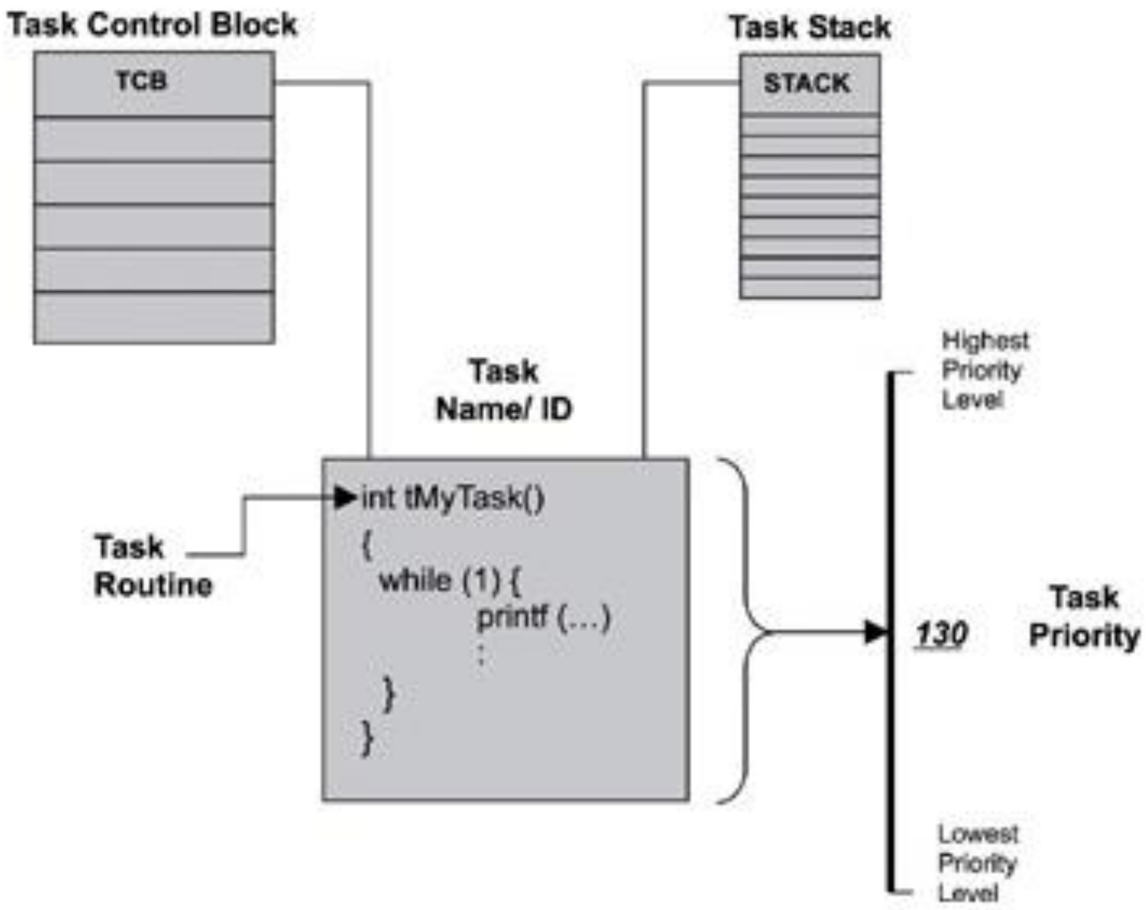
# Content

- Task
- Task synchronization
  - Semaphore and mutex
  - Philosopher’s problem
- Deadlock

# Defining a task

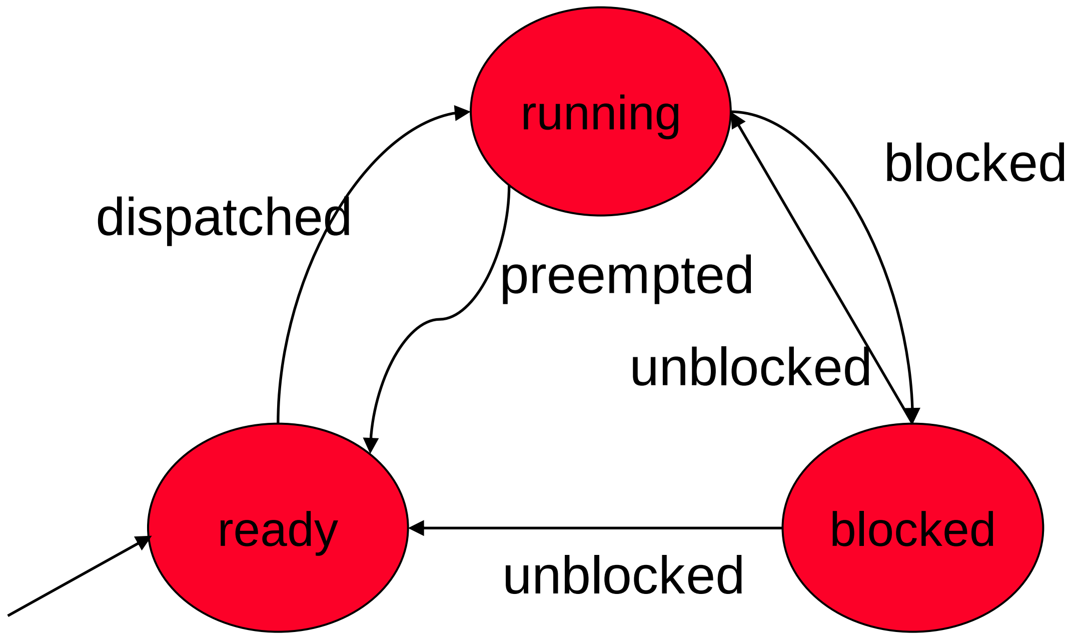
- Independent thread of execution that can compete with concurrent tasks for processor execution time.
  - Thus, schedulable & allocated a priority level according to the scheduling algorithm
- Elements of a task
  - Unique ID
  - Task control block (TCB)
  - Stack
  - Priority (if part of a preemptive scheduling plan)
  - Task routine

# Elements of a task



# Task states & scheduling

- Task states:
  - Ready state
  - Running state
  - Blocked state
- Scheduler determines each task's state.



# Task states

- **ready state**-the task is ready to run but cannot because a higher priority task is executing.
- **blocked state**-the task has requested a resource that is not available, has requested to wait until some event occurs, or has delayed itself for some duration.
- **running state**-the task is the highest priority task and is running.

# Typical task structure(1)

- Typical task structures:
  - Run-to-completion task: Useful for initialization & startup tasks

```
RunToCompletionTask ()
{
        Initialize application
        Create 'endless loop tasks'
        Create kernel objects
        Delete or suspend this task
}
```

# Typical task structure(2)

- Typical task structures:
  - Endless-loop task: Work in an application by handling inputs & outputs

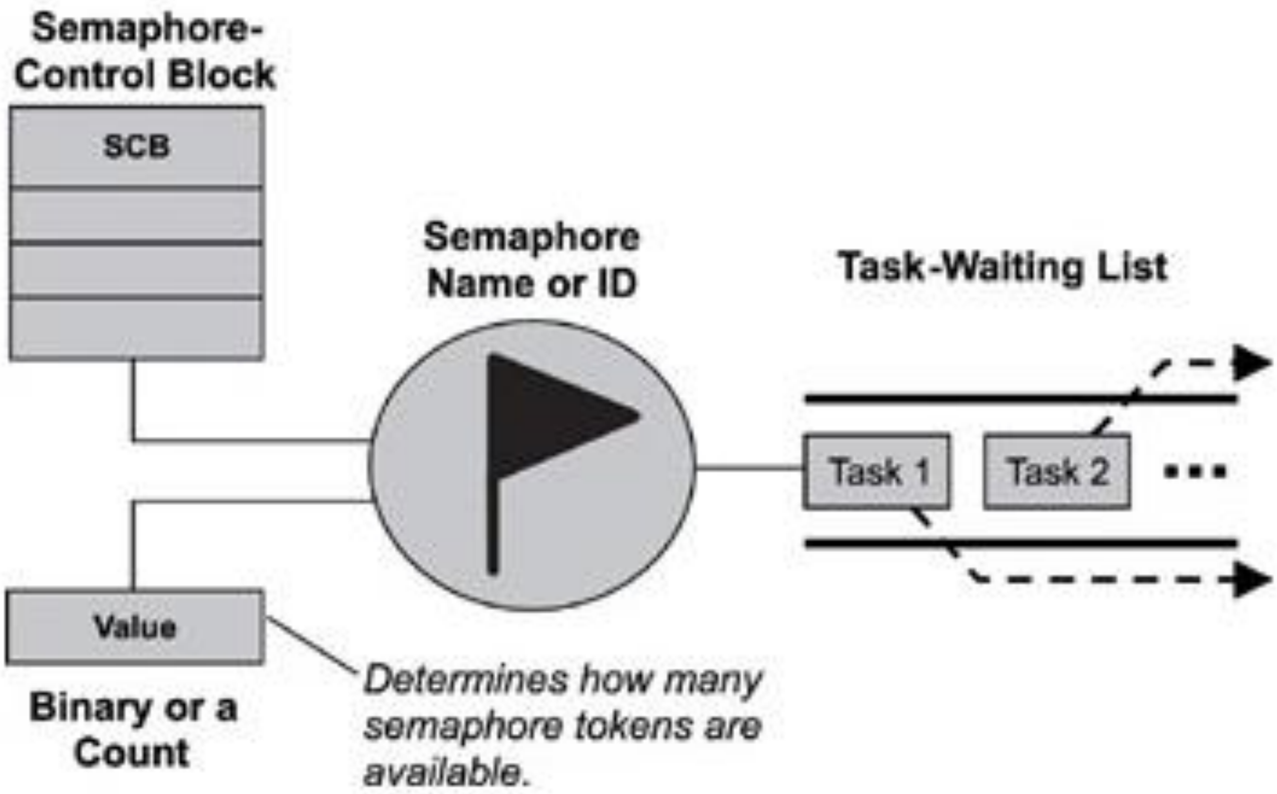
```
EndlessLoopTask ()
{
        Initialization code
        Loop Forever
        {
                Body of loop
                Make one or more blocking calls
        }
}
```

# Semaphores

- In multi-task systems, concurrently-running tasks should be able to
  - synchronize their execution, and
  - to coordinate mutual exclusive access to shared resources.

- What is semaphore?
  - A kernel object to realize synchronization & mutual exclusion
  - One or more threads of execution can acquire & release to execute an operation to be synchronized or to access to a shared resource.

# Semaphore elements



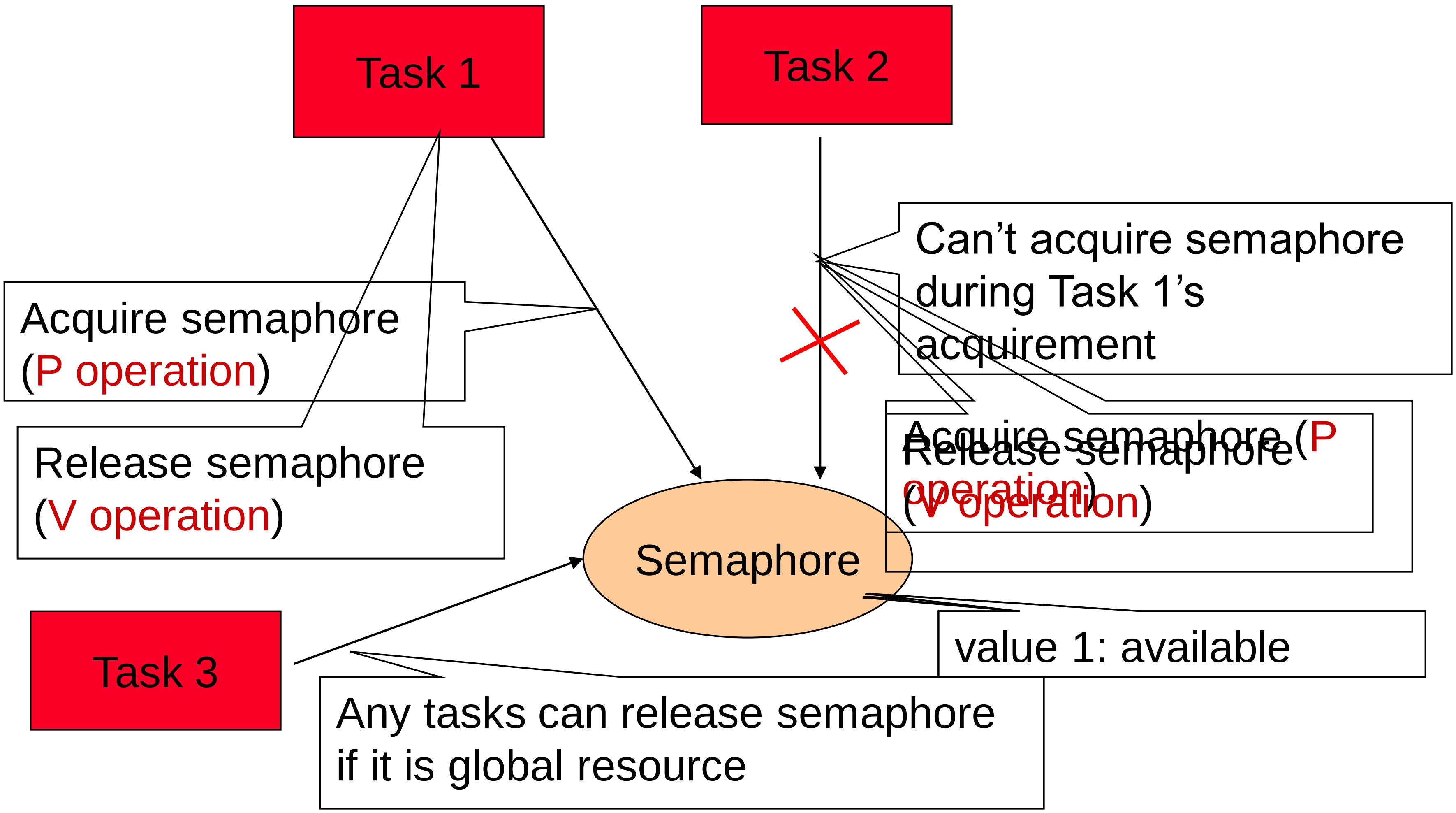
# Defining semaphore

- Elements of semaphores assigned by the kernel
  - Semaphore control block (SCB)
  - Semaphore ID (unique in the system)
  - Value (binary or count)
  - Task-waiting list

- Classification of semaphores:
  - Binary semaphore
  - Counting semaphore
  - Mutex

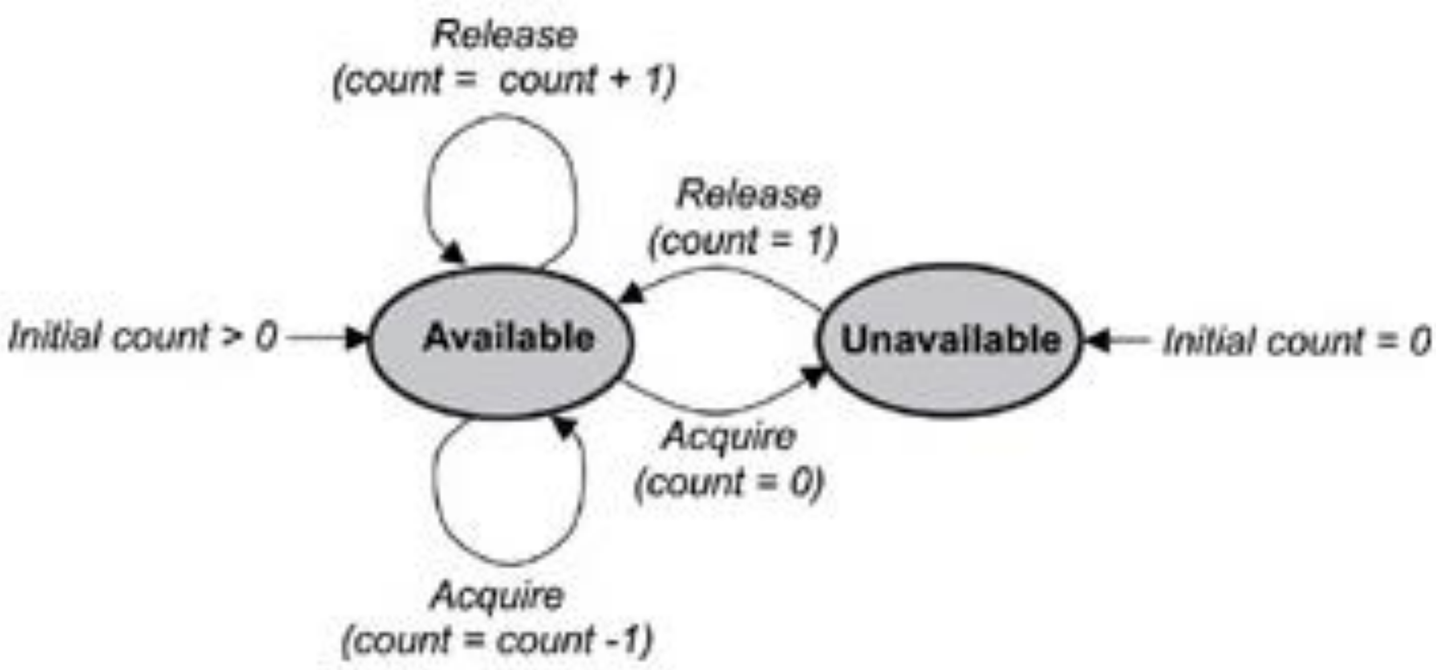
# Binary semaphore

- Provides binary state of unavailable/available (or empty/full)



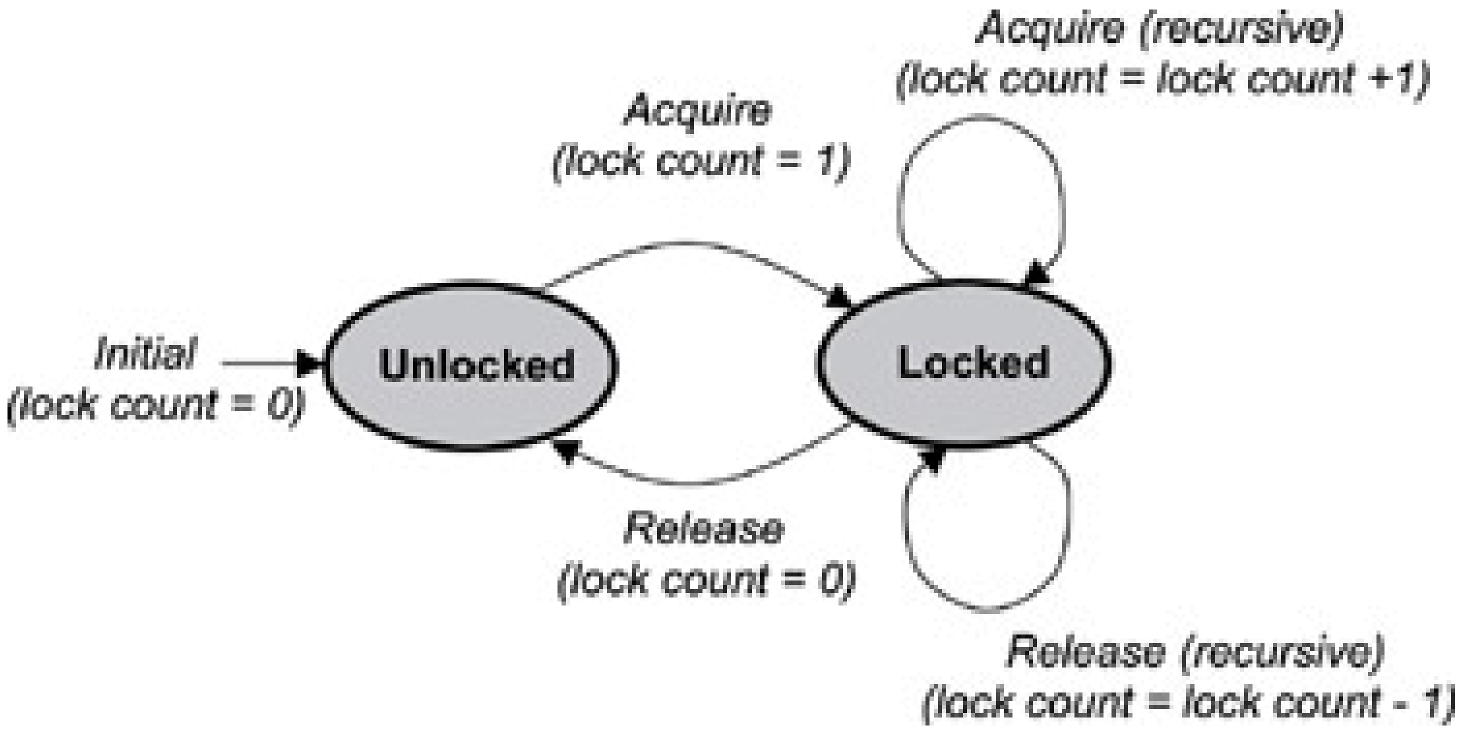
# Counting semaphore

- If the value of a semaphore is *n*, it can be acquired *n* times concurrently.



# Mutual exclusion (MUTEX) semaphores

- Special binary semaphore
- Has states of locked/unlocked & lock count
- Difference: signaling vs protecting



# Semaphore vs Mutex


All processes can release semaphore



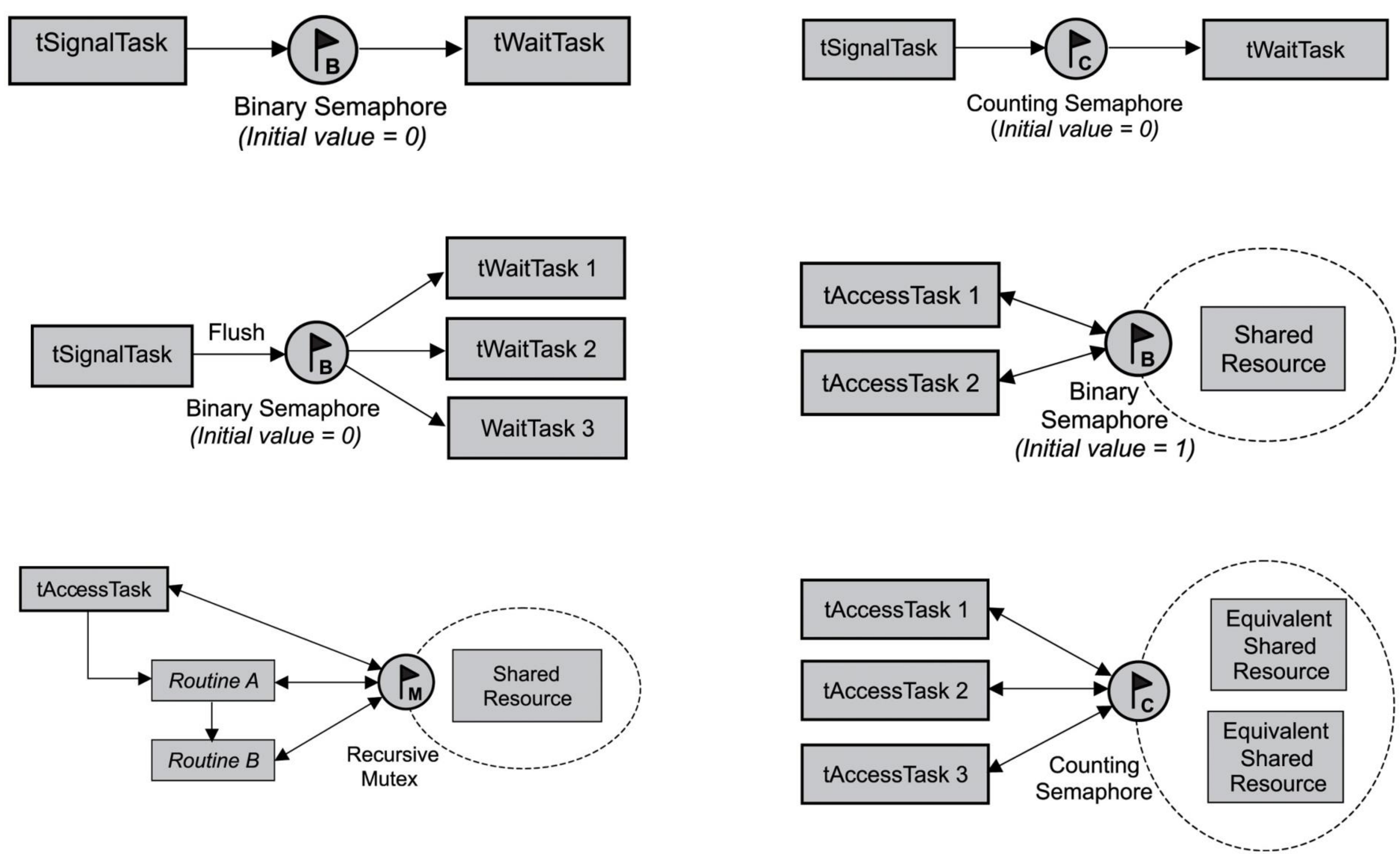
Only owner can unlock mutex

# Typical semaphore use

- Semaphore are used for:
  - Synchronizing execution of tasks
  - Coordinating access to a shared resource

- Synchronization design requirements
  - Wait-and-signal
  - Multiple-task wait-and-signal
  - Credit-tracking
  - Single shared-resource-access
  - Recursive shared-resource-access
  - Multiple shared-resource access

# Typical semaphore uses



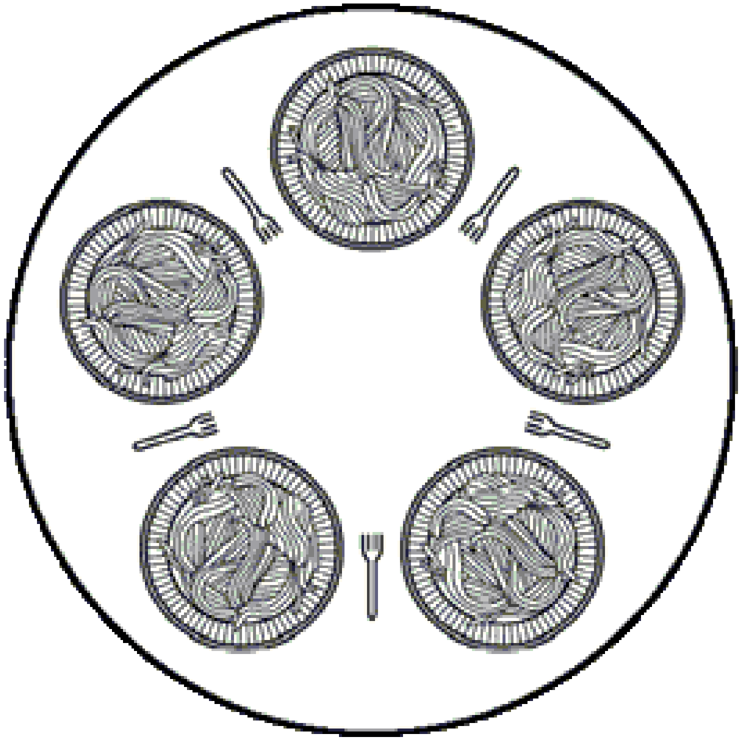
# Example:

- Consumer – producer problem
- Producer:
  - Task to read data from input device,
  - Data transferred to 4KB shared buffer memory
- Consumer:
  - Task to read and process data from buffer memory

➔ Synchronization problem

- Solution 1: binary semaphore for buffer memory access
- Other solution?

# Example: The Dining Philosophers Problem



- Five philosophers seated around a circular table with a plate of spaghetti for each.
- Between each pair of plates is one fork
- The spaghetti is so slippery that a philosopher needs two forks to eat it.
- When a philosopher gets hungry, he tries to acquire his left and right fork

Problem: not enough forks for all

➔ Write a program to control philosophers concurrently without getting stuck?

# Solution 1:

```
#define N 5/* number of philosophers */

void philosopher(int i)/* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( ); /* philosopher is thinking */
        take_fork(i); /* take left fork */
        take_fork((i+1) % N);/* take right fork; */
        eat(); /* yum-yum, spaghetti */
        put_fork(i); /* Put left fork back on the table */
        put_fork((i+1) % N);/* put right fork back table */
    }
}
```

This is non-solution, because of potential deadlock. Why?

# Solution 2:

- If philosopher could not acquire fork:
  - Wait for a random duration
  - Retry
- Simple and efficient
- Minimize lock-state time
- But not lock-state free

➔ Cannot be used in critical system

# Solution 3

```
#define N 5 /* Number of philosphers */

#define RIGHT(i) (((i)+1) %N)

#define LEFT(i) (((i)==N) ? 0 : (i)+1)

typedef enum { THINKING, HUNGRY, EATING } phil_state;

phil_state state[N];

semaphore mutex =1;

semaphore s[N]; /* one per philosopher, all 0 */

void test(int i) {

  if ( state[i] == HUNGRY && state[LEFT(i)]!= EATING &&
  state[RIGHT(i)] != EATING )

  {

       state[i] = EATING;

       up(s[i])

  }

}
```

# Solution 3

```
void get_forks(int i){
   down(mutex);
   state[i] = HUNGRY;
   test(i);
   up(mutex);
   down(s[i]); //lock
}
void put_forks(int i){
   down(mutex);
   state[i]= THINKING;
   test(LEFT(i));
   test(RIGHT(i));
   up(mutex);
}
```

```
void philosopher(int process)
{
    while(1)
    {
        think();
        get_forks(process);
        eat();
        put_forks(process);
    }
}
```

# Deadlock

- Task uses resources
- Two types of resources
  - Preemtible: memory, printer
  - Non-preemtible: CD-W drive in disc burning process
- Potential deadlock with preemtible resources can be resolved (imagine the case philosophers can share forks without cleaning!)
- Deadlock involves non-preemtible resources

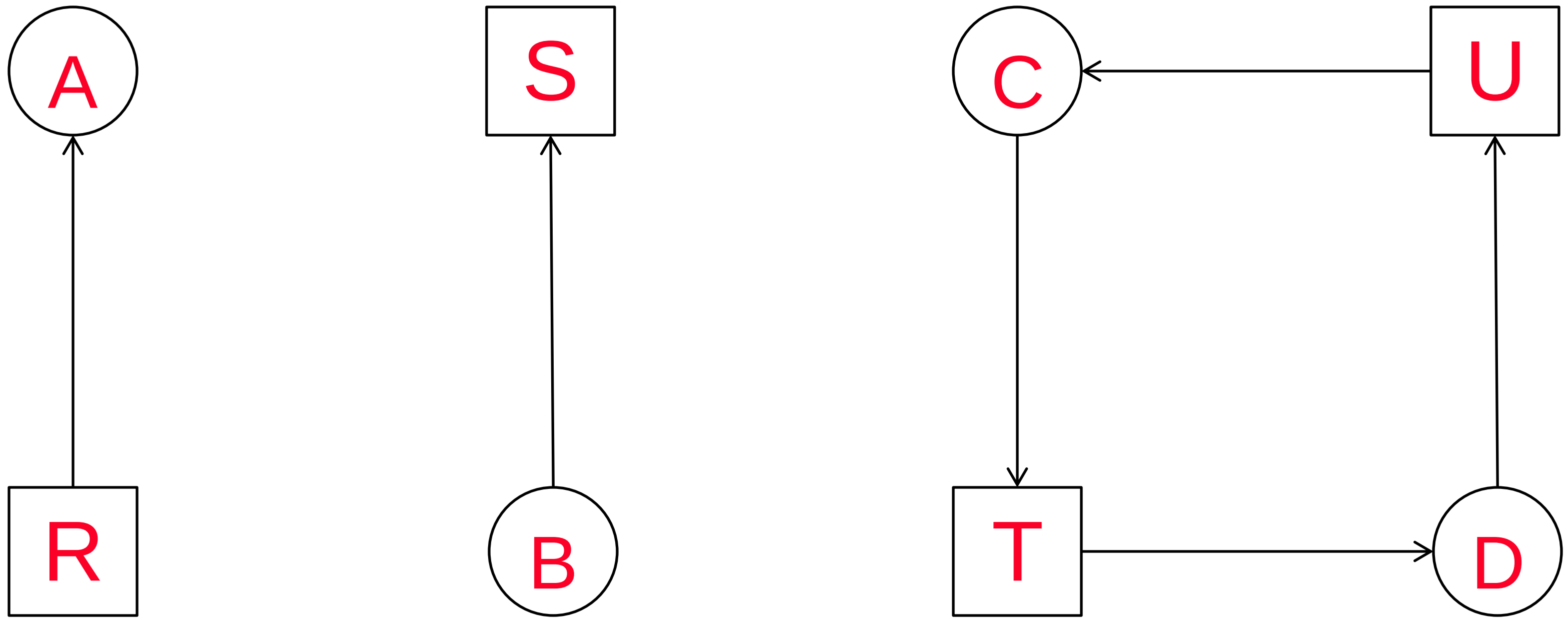
# Deadlock

- Deadlock definition:

  *A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the process can occur.*

- Conditions for deadlock
  - Mutual exclusion condition: a resource that cannot be used by more than one process at a time
  - Hold and wait condition: processes already holding resources may request new resources
  - No preemption condition: No resource can be forcibly removed from a process holding it, resources can be released only by the explicit action of the process
  - Circular wait condition: two or more processes form a circular chain where each process waits for a resource that the next process in the chain holds

# Deadlock modeling




Process A holds resource R

Process B acquires resource R

Deadlock
Process C is waiting resource T, which is currently holding by process D.
Process D is waiting resource U, which is currently holding by process C.

# Deadlock detection and recovery

- Let deadlock occurs, tries to detect and attempt recovery if necessary.
- 2 methods to detect deadlock
  - Deadlock detection with one resource of each type
  - Deadlock detection with multiple resource of each type
- 3 methods to recovery from deadlock
  - Recovery through preemption
  - Recovery through rollback
  - Recovery through killing processes

# Deadlock detection with one resource

- Suppose that system has only one resource for each type such as 1 printer, 1 tape driver, 1 plotter….
- To detect a deadlock (the easiest technique)
  - Draw a graph of relationship between processes and resources
  - If at least one cycle can be detected, a deadlock exists.

# Example

- Process A holds R and wants S
- Process B holds nothing but wants T
- Process C holds nothing but wants S
- Process D holds U and wants S and T
- Process E holds T and wants V
- Process F holds W and wants S
- Process G holds V and wants U