# CS343: Operating System

# Threading and Synchronization

## Lect18 : 08th Sept 2023

**Dr. A. Sahu**

**Dept of Comp. Sc. & Engg.**

**Indian Institute of Technology Guwahati**

# Outline

- Threading
- Threading Examples
- Thread mappings
  - Pthread/Uthread, Kthread, Hthread
- Synchronization

# Posix Threads (Pthreads) Interface

- **Creating and reaping threads**
  - `pthread_create, pthread_join`
- **Determining your thread ID :** `pthread_self`
- **Terminating threads**
  - `pthread_cancel, pthread_exit`
  - `exit` [terminates all threads] , `return` [terminates current thread]
- **Synchronizing access to shared variables**
  - `pthread_mutex_init, pthread_mutex_[un]lock`
  - `pthread_cond_init, pthread_cond_[timed]wait`

# User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads,** Windows threads, Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows , Solaris, Linux

# User Level Thread

- **Advantages**
  - User level threads are simpler and faster to generate. They are also easier to manage.
  - Thread switching in user-level threads doesn't need kernel mode privileges.
  - These are more portable.
  - These threads may be run on any OS.
- **Disadvantages**
  - The complete process is blocked if a user-level thread runs a blocking operation.
  - User-level threads don't support system-wide scheduling priorities.
  - It is not appropriate for a multiprocessor system.

# Kernel Level Thread

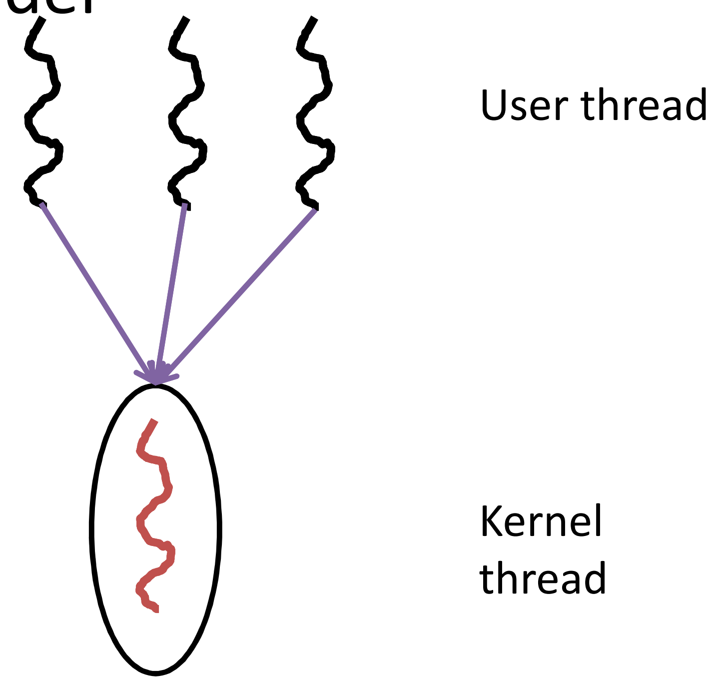
- **Advantages**
  - If a thread in the kernel is blocked, it does not block all other threads in the same process.
  - Several threads of the same process might be scheduled on different CPUs in kernel-level threading.
  - Kernel routines can be multithreaded as well.
- **Disadvantages**
  - Compared to user-level threads, kernel-level threads take longer to create and maintain.
  - A mode switch to kernel mode is important to transfer control from one thread in a process to another.

# Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

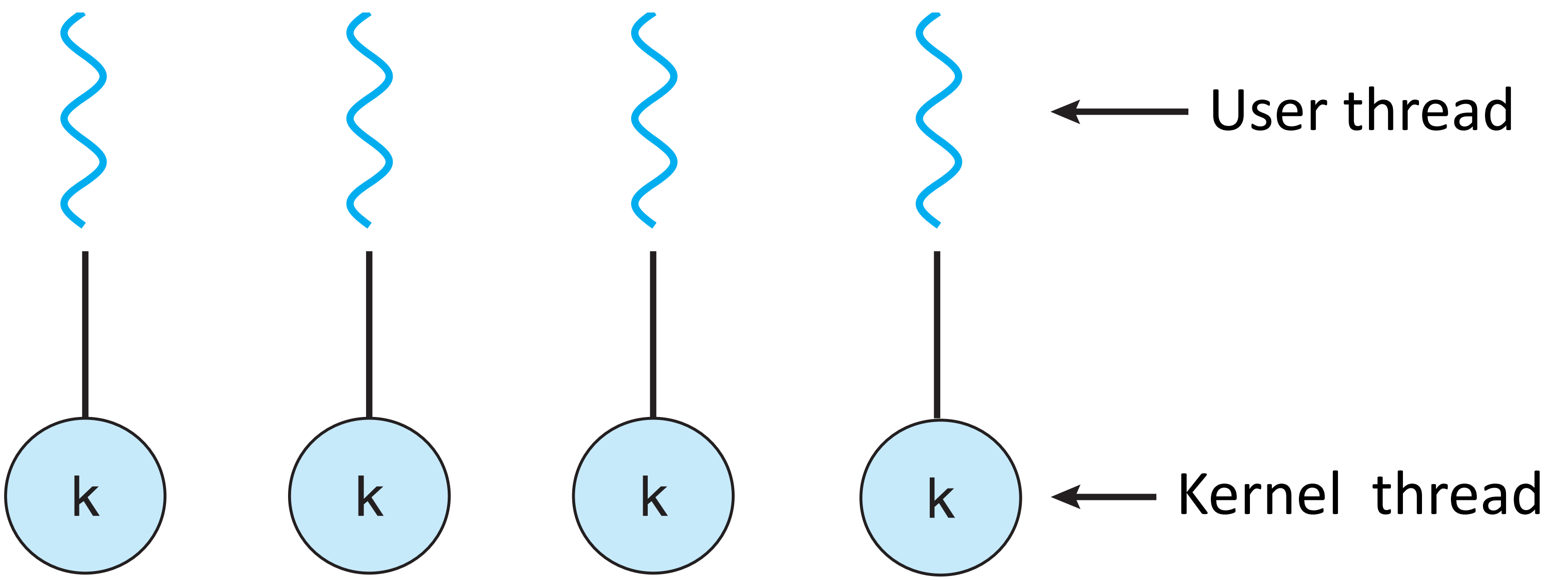
# Many-to-One

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



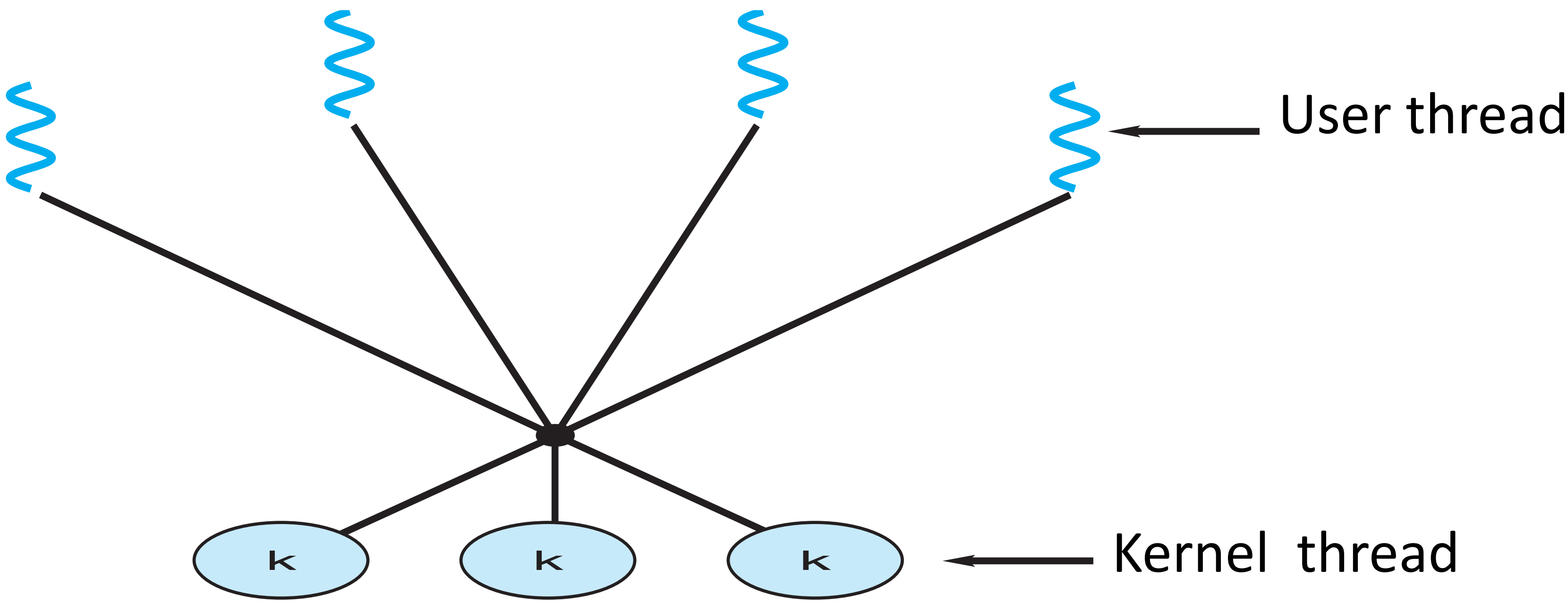
# One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples: Windows, Linux, Solaris 9 and later



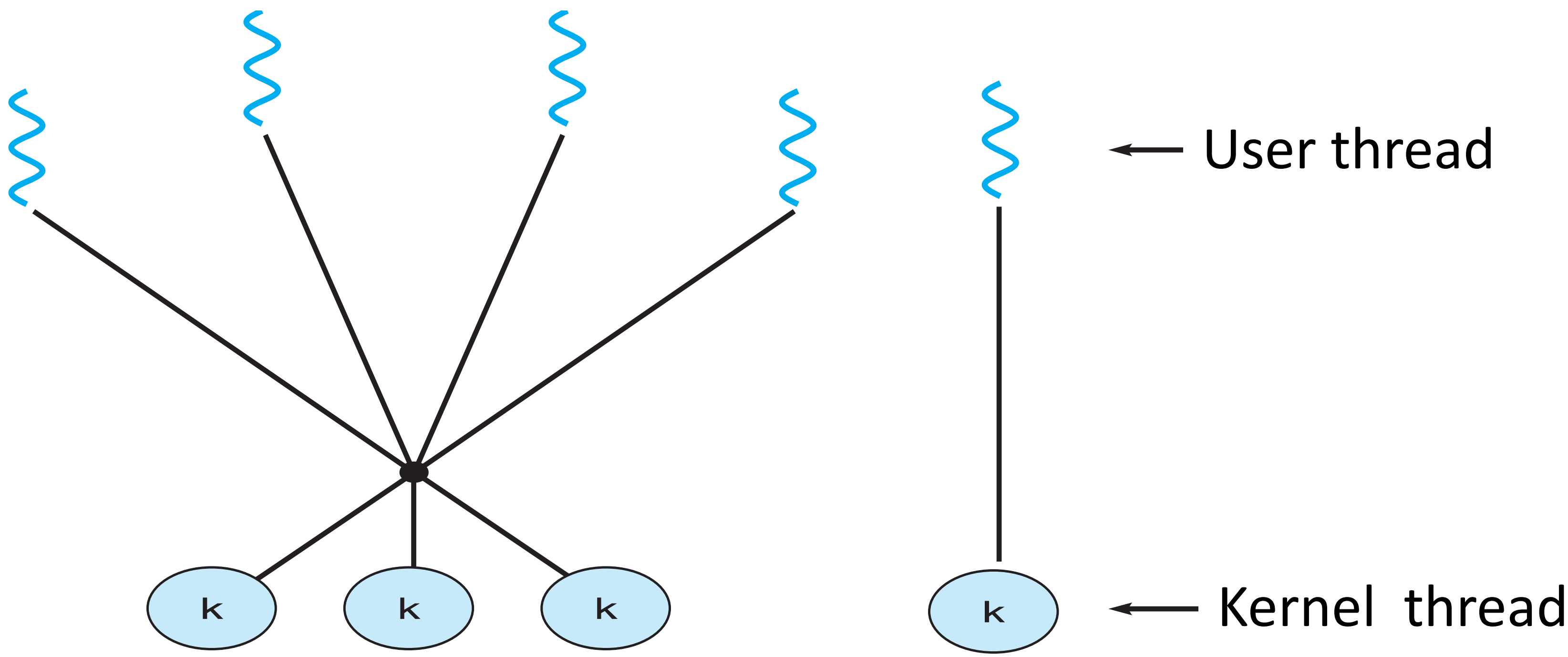
# Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the  operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows  with the *ThreadFiber* package



# Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - IRIX, HP-UX, Tru64 UNIX
  - Solaris 8 and earlier



# Multithreaded program on Multi-core

- Uthread, Kthread
- Hthread (hardware thread) : **MIT term HART**
  - **Provided by hardware CPU**
- Mapping Kthread to Hthread (Same model)
- Mapping Uthread to Hthread (By passing Kthread) HART
  - ***Allow creation an Uthread if a Hthread is free***
- **Thread affinity**
  - Suppose a thread is suitable to a particular Hthread
  - Benefit of cache and resources

# Running a Apps on Specific core

```
#define _GNU_SOURCE
#include <pthread.h>  #include <stdio.h> #include <stdlib.h>  #include <errno.h>
#define handle_error_en(en, msg) \
        do { errno = en; perror(msg); exit(EXIT_FAILURE); } while (0)
int MYBenchmark() {  while(1); }

int main(int argc, char *argv[]){    // a.out 3 # run the apps on core id =3
        int Proc,s;                  // Check on System Monitor  Apps
        cpu_set_t  cpuset;
        pthread_t   thread;
        thread = pthread_self();
        Proc=atoi(argv[1]);
        CPU_ZERO(&cpuset);
        CPU_SET(Proc, &cpuset);
        s = pthread_setaffinity_np(thread, sizeof(cpu_set_t), &cpuset);
        MYBenchmark();
        exit(EXIT_SUCCESS);
}
```

# Shared Variable

```
#define NTH 10
int counter=0;//Shared among thread
void  SimpleCnt() {
  for(int i=0;i<10;i++)
     counter++;
}
```

# Shared Variable: SimpleCnt

```
int main() {
  thread_t th[NTH];
  int i, j;
  for(i=0; i < NTH; i++) {

  thread_create(&th[i],NULL,SimpleCnt,NULL
  );
  }
  for(j=0; j < NTH; j++) {
     pthread_join( th[j], NULL);
  }
  printf("Final Ctr val: %d\n", counter);
}
```

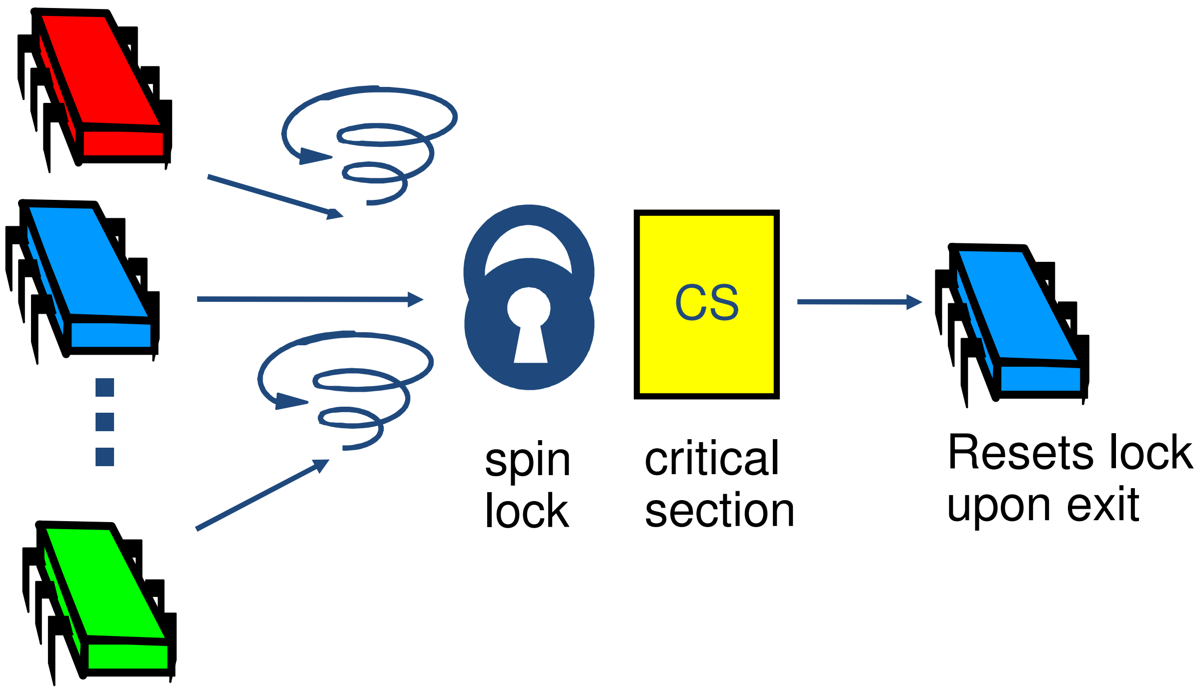
# Shared Variable with Mutex

```
#define NTH 10
pthread_mutex_t M = 0;
void  MutexCnt() {
  for(int i=0;i<10;i++){
     pthread_mutex_lock( &M );
     counter++;
     pthread_mutex_unlock( &M);
  }
}
```

# Shared Variable: MutexCnt

```
int  counter = 0;
int main() {
  thread_t th[NTH];
  int i, j;
  for(i=0; i < NTH; i++) {
    thread_create(&th[i],NULL,MutexCnt,NULL);
  }
  for(j=0; j < NTH; j++) {
     pthread_join( th[j], NULL);
  }
  printf("Final Ctr val: %d\n", counter);
}
```

# Many threads trying to acquire Mutex LOCK



# Shared Variable with Mutex

```
#define NTH 10
pthread_mutex_t M = 0;
void  MutexCnt() {
 int local_cnt=0
  for(int i=0;i<10;i++)local_cnt++;
    pthread_mutex_lock( &M );
    counter +=local_count;
    pthread_mutex_unlock( &M);
}
```

# Synch. Primitives

- pthread_mutex_init, lock, unlock, trylock
- pthread_attr_set detachstate, guardsize_np, stacksize, inheritsched, schedpolicy, schedparam
- pthread_cond_wait, signal, broadcast, init, destroy
- Our main concern
  - Lock, unlock, trylock, condsignal, condbroadcast

# Locking Overhead

- Serialization points
  - Minimize the size of critical sections
  - Be careful
- Rather than wait, check if lock is available
  - **pthread_mutex_trylock**
  - If already locked, will return EBUSY
  - Will require restructuring of code
    - **Suspend self by pthread_yeild() Give chance to others**
    - **Suspend self by doing a timed wait..**

1. Process and Thread Interchangeably
Processes/Threads share data

2. Mutex Via Semaphore

# Semaphore

- Earlier : Shared thread incrementing counter
- Train : One track many trains
  - **MUT**ual **EX**clusion is must
  - **Train Collision**

Life loss

# Semaphore

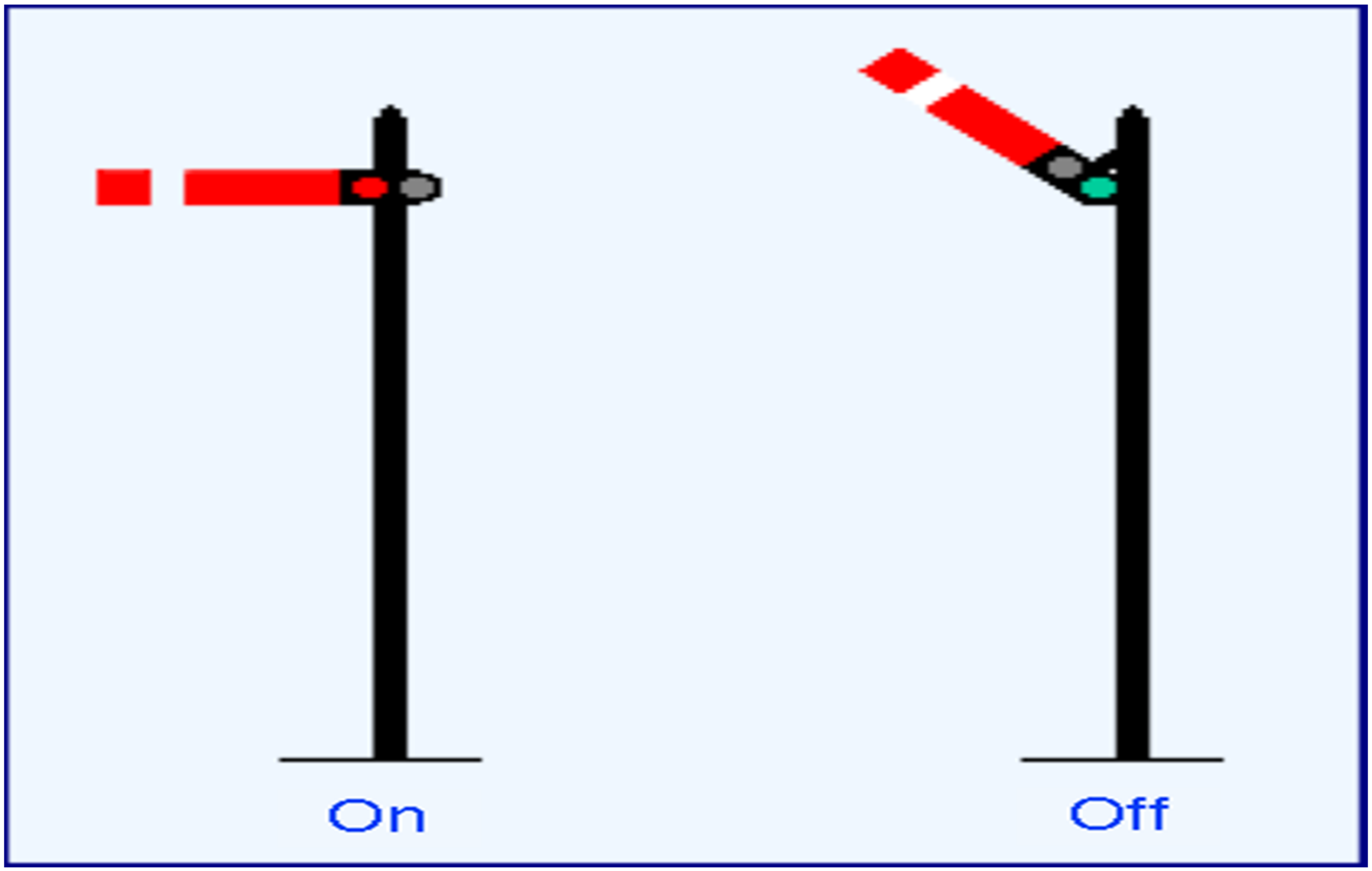
- Semaphore : A  system of sending messages by holding  The arms or two flags or poles in certain positions according to an alphabetic code

# Semaphore

- Semaphore : A system of sending messages by holding The arms or two flags or poles in certain positions according to an alphabetic code

# Semaphore

- Semaphore : A  system of sending messages by holding The arms or two flags or poles in certain positions according to an alphabetic code



# Locking Overhead

- Serialization points
  - Minimize the size of critical sections
  - Be careful
- Rather than wait, check if lock is available
  - **pthread_mutex_trylock**
  - If already locked, will return EBUSY
  - Will require restructuring of code
    - **Suspend self by  pthread_yeild() Give chance to others**
    - **Suspend self  by doing a timed wait..**

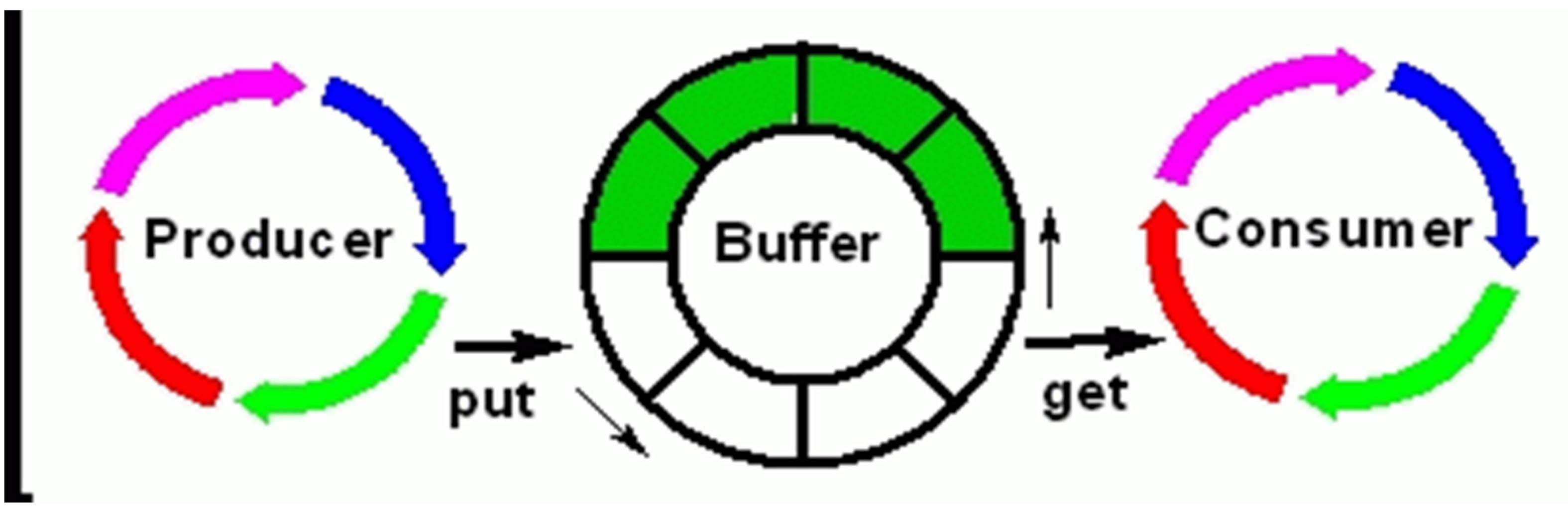
# Synchronization Hardware and Algorithms

# Data Consistency

- Processes/thread can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

# Classic Example: Producer & Consumer

- There is a buffer of size  BUFFER_SIZE
- When a producer : produce an Item,  he put into Buffer and increment the counter
- When a consumer : consume an Item,  he read an Item from Buffer and decrement the counter
- Initially, **counter** is set to 0



# Producer

```
while (true) {
        /* produce an item in next produced */
        while (counter == BUFFER_SIZE) ;
                /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

# Consumer

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        /* consume the item in next consumed */
}
```

# Assumption

- Assume that all the instruction are atomic
  - The **load** and **store** machine-language instructions are atomic;
  - That is, cannot be interrupted
- We will see: Still we have problem in Synchronization
  - Or some protocol/Algorithm to Handle
  - We may need different hardware support, **a specific kind of Instruction to be atomic**

# Race Condition: ProduserConsumer

**Counter ++**
could be implemented as

**reg1 = counter //LD**
**reg1 = reg1 + 1 //INC**
**counter = reg1 //ST**

**Counter - -**
could be implemented as

**reg2 = counter //LD**
**reg2 = reg2 – 1 //DEC**
**counter = reg2 //ST**

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

S0: producer execute **reg1 = counter** {reg1 = 5}
S1: producer execute **reg1 = reg1 + 1** {reg1 = 6}
S2: consumer execute **reg2 = counter** {reg2 = 5}
S3: consumer execute **reg2 = reg2 – 1** {reg2 = 4}
S4: producer execute **counter = reg1** {counter = 6 }
S5: consumer execute **counter = reg2** {counter = 4}

**Expected Result =5**

# Critical Section Problem

- Consider system of ***n*** processes

  $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section

# Critical Section Problem

- ***Critical section problem*** is to design protocol to solve this
  - Each process must ask permission to enter critical section in **entry section**,
  - May follow critical section with **exit section**, then **remainder section**

# Critical Section

- General structure of process ***P****$_i$*

```
do {
        entry section
                critical section
        exit section
                reminder section
} while (true)
```

Lock ()

Unlock ()

# Thanks