Operating System Sessional
POSIX THREAD - pthread

CSE-308

(SEC-A)

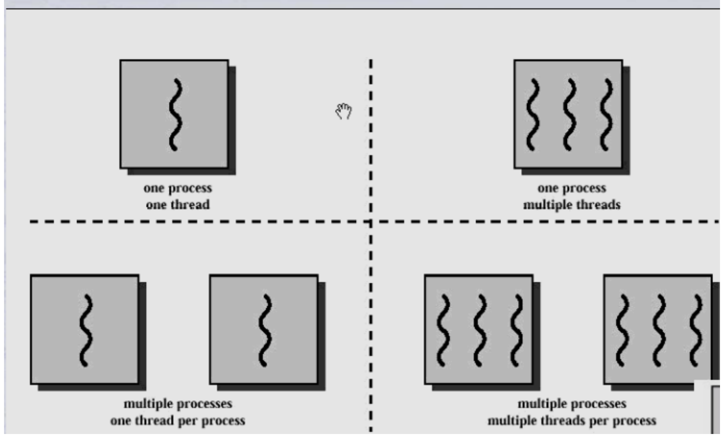
Lec Tasfia Sara, CSE
Spring 2024, CSE-22

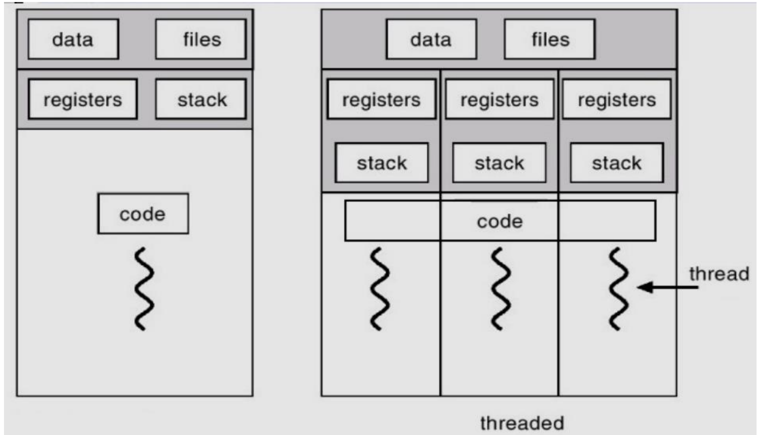
Topics

- **pthread**
  - **pthread Introduction**
  - **Process vs Thread**
  - **Create Thread**
  - **Pass arguments**
  - **Join Thread**
  - **Critical Region**
  - **Lock**
  - **Try Lock**
  - **Example/ Simulation**

Thread Recap

1. A thread is a light weight process. Every thread belongs to exactly one process.
2. A thread is **a single sequence stream within a process**.
3. There is no need to allocate extra memory space for threads.
4. Each thread has **its own program counter (PC), a register set, and a stack space**.





Thread Recap

5. Threads shares with other threads' their code section, data section and OS resources like open files and signals within the same process.

6. For switching one thread to another **no need to interact with operating system, interacts with process only.**

7. Threads are not independent from each other unlike processes. For example,

- A failure in one thread (e.g., accessing invalid memory) can potentially crash the entire process, affecting all threads within it.
- If one thread modifies a shared variable, this change is immediately visible to other threads.

Thread Recap

Threads are popular way to improve application through **parallelism**. For example, in a browser, multiple tabs can be different threads. MS word uses multiple threads, one thread to format the text, other thread to process inputs, etc.

Threads operate faster than processes due to following reasons:

1) Thread creation is much faster

2) **Context switching between threads is much faster**

3) Threads can be terminated easily

4) **Communication between threads is faster**

Parallelism is the practice of dividing a task into smaller sub-tasks that can be processed simultaneously to improve performance and efficiency.

Pthread

In computing, POSIX (Portable Operating System Interface) Threads, commonly known as pthreads, is an execution model that exists independently from a programming language, as well as a parallel execution model.

1. It allows a program **to control multiple different flows of work** that overlap in time.
2. Pthreads are used to leverage the power of multiple processors.
3. Here a process is broken into threads, each thread can use a processor to complete its execution and because there are multiple processors executing threads at the same time, parallelism in execution can be seen.

Pthread

- Each flow of work is referred to **as a thread**, and creation and control over these flows is achieved by making calls to the POSIX Threads API.
- POSIX Threads is an API (Application Programming Interface) defined by the Institute of Electrical and Electronics Engineers (IEEE) standard POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995).
- The POSIX Threads API includes functions for **creating, controlling, and synchronizing** threads.
- POSIX (Portable Operating System Interface) is a set of standard operating system interfaces based on the Unix operating system.

pthread Syntax: Thread Creating

- Create a pthread Instance
  - `pthread_t obj;`
- Start Executing Thread
  - **pthread_create function will create a kernel level thread**
  - **To mention the thread fn and fn args**
  - `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start)(void *), void *arg);`
- Join Thread or Wait For Thread to Finish
  - **pthread_join will wait for a particular thread**
  - **To catch the return**
  - `int pthread_join(pthread_t thread, void **status);`

Sleep Function (Yielding)

- **Gives chances to others**
- **Sleep(int msec);**
- **uSleep(int usec);**

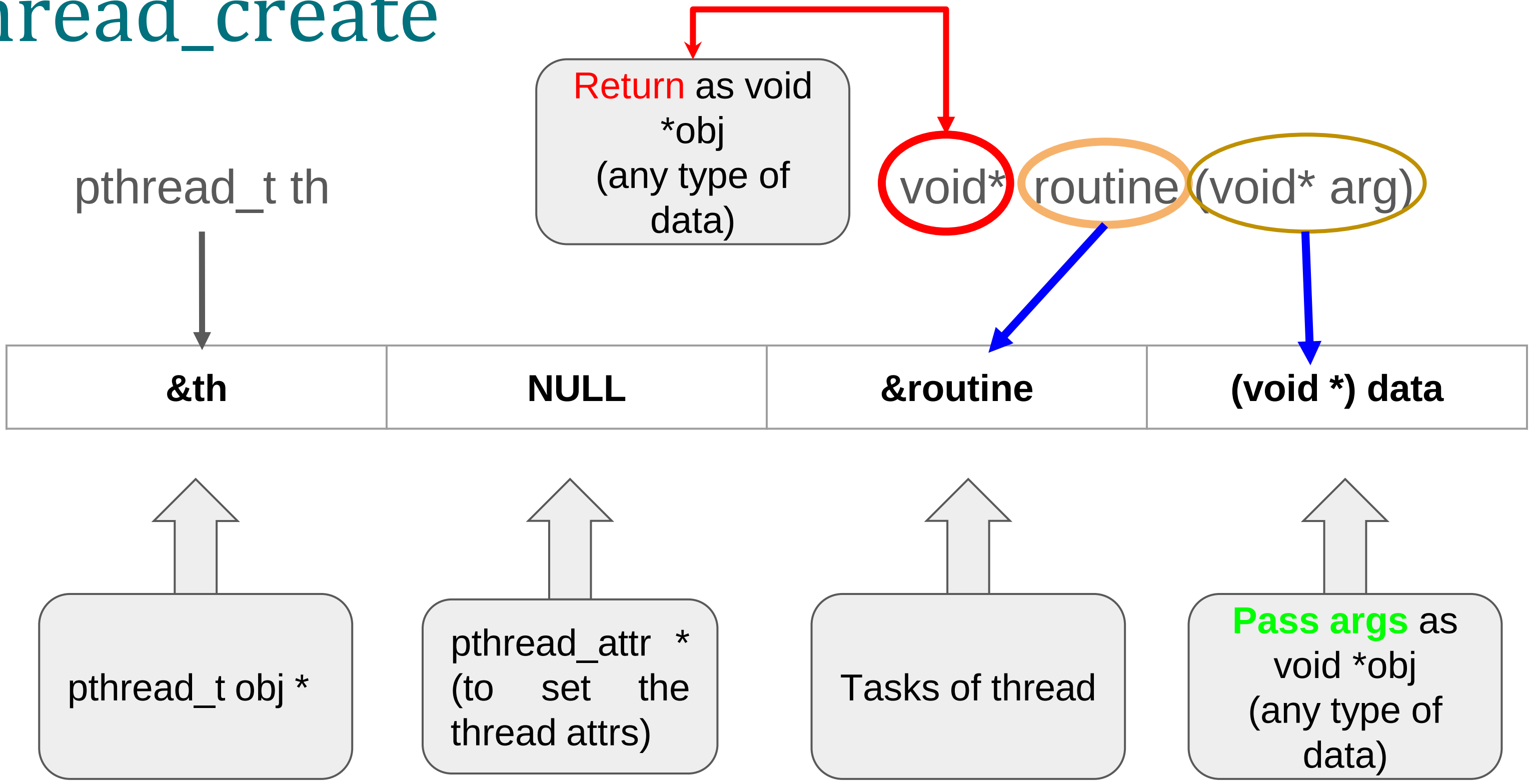
```
12:38:03 My turn!
12:38:04 Your turn!
12:38:04 My turn!
12:38:05 My turn!
12:38:06 Your turn!
12:38:06 My turn!
12:38:07 My turn!
12:38:08 Your turn!

Process returned 0 (0x0)   execution time : 6.044 s
Press any key to continue.
```

```
12:39:57 My turn!
12:39:58 Your turn!
12:39:58 My turn!
12:39:59 My turn!
12:40:00 Your turn!
12:40:00 My turn!
12:40:01 My turn!
12:40:02 Your turn!
12:40:02 My turn!
12:40:03 My turn!
12:40:04 My turn!
Both threads have finished execution.

Process returned 0 (0x0)   execution time : 8.089 s
Press any key to continue.
```

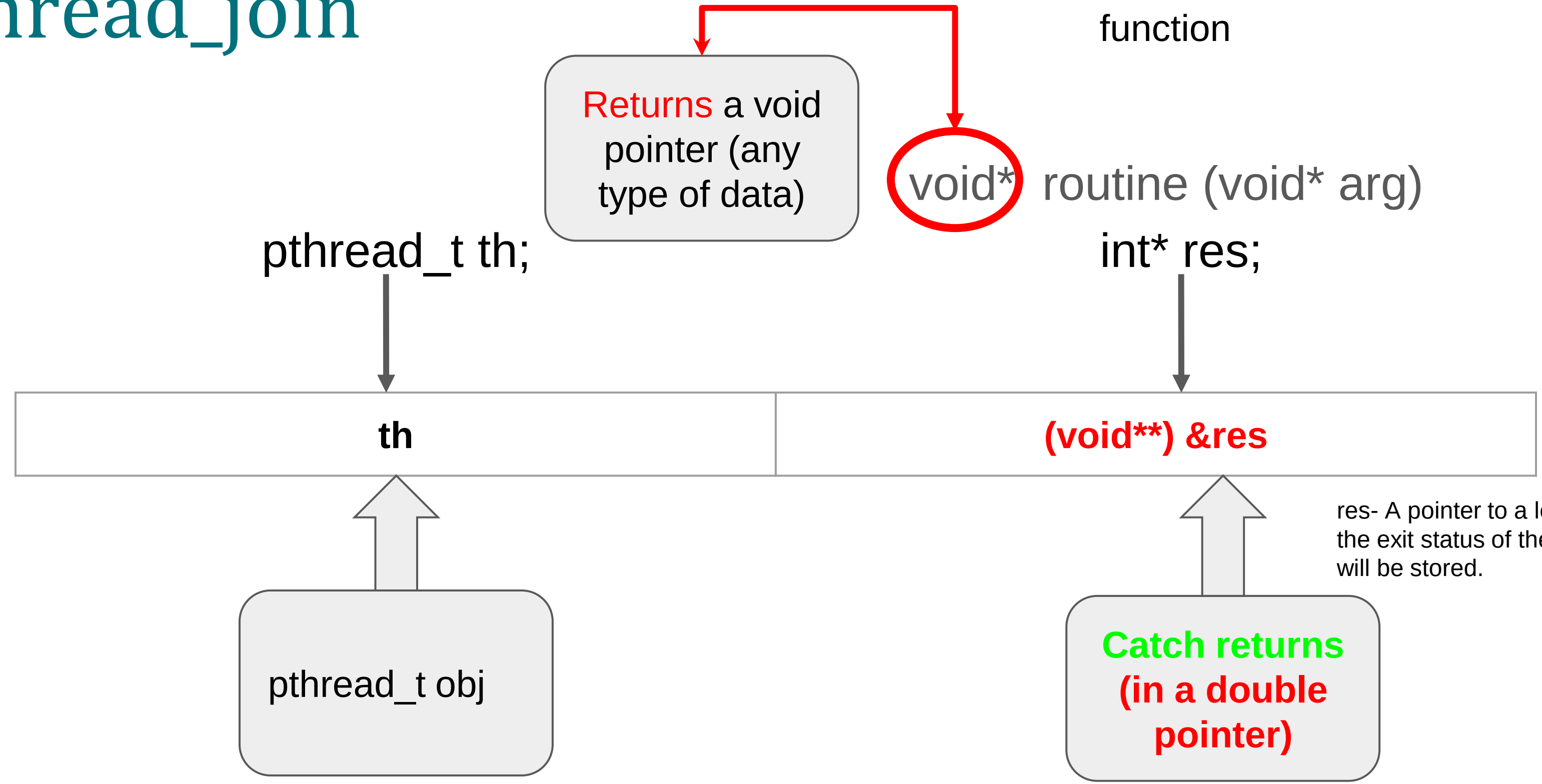
Pthread_create



We can pass a pointer to a `pthread_attr_t` structure to customize attributes like stack size, scheduling policy, etc.,

Pthread_join

Double pointer is used to capture the return value of the thread function



res- A pointer to a location where the exit status of the joined thread will be stored.

When you call `pthread_join` on a thread, it effectively blocks the calling thread (usually the main thread) until the specified thread has terminated. This is particularly useful when you need to ensure that a thread has completed its work before proceeding with the rest of the program.

Threads and Critical Section

Thousands of threads performing the same routine function.

Though the instructions are same (they share the same address space), their context are not the same.

A critical section is a part of the code where shared resources (like std::cout) are accessed. As they share the standard output (std::cout), the terminal. Which means it's the critical section!!

Since all threads are accessing std::cout at the same time, their outputs can get mixed up. For example, one thread might print "Hello" while another is in the middle of printing "Ending thread".

```
void* routine(void *args) {
    cout<<"Hello from threads"<<endl;
    Sleep(3);
    cout<<"Ending thread"<<endl;
}
```

```
pthread_t th[1000];
int i;
for (i = 0; i < 1000; i++) {
    if (pthread_create(&th[i], NULL, &routine, NULL) != 0) {
        perror("Failed to create thread");
        return 1;
    }
    ///printf("Thread %d has started\n", i);
}
```

Critical Section

INC in x86 is not atomic!!
(means it can be interrupted)

But x may be updated in T2.
Say x contains 1000, but T1 uses the old value.

```
int x = 2;

void* routine(void *args) {
    for (int i=0; i<100000;i++){
        x += 5;
        ///Sleep(1);
        ///printf("Value of x: %d\n", x);
    }
}
```

Critical Section

Context Switch in the middle of the instruction (as it's not an atomic instruction)

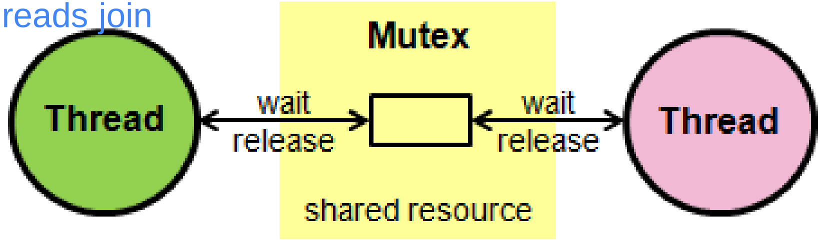
Instruction	T1	
Read X	Done	AX = 505
Context Switch		
Add 5	Resume	AX = 510
Write x		x = 510

pthread Syntax: Thread Synchronization

- Critical Section
  - The critical section refers to the segment of code where processes access shared resources, such as common variables and files, and perform write operations on them.
- Mutex
  - Mutex (mutual exclusion object) is a program object that is created so that multiple program thread can take turns sharing the same resource, such as access to a file.
- Try Mutex
  - Same as Mutex but instead of waiting it returns 0.
- Conditional Variable

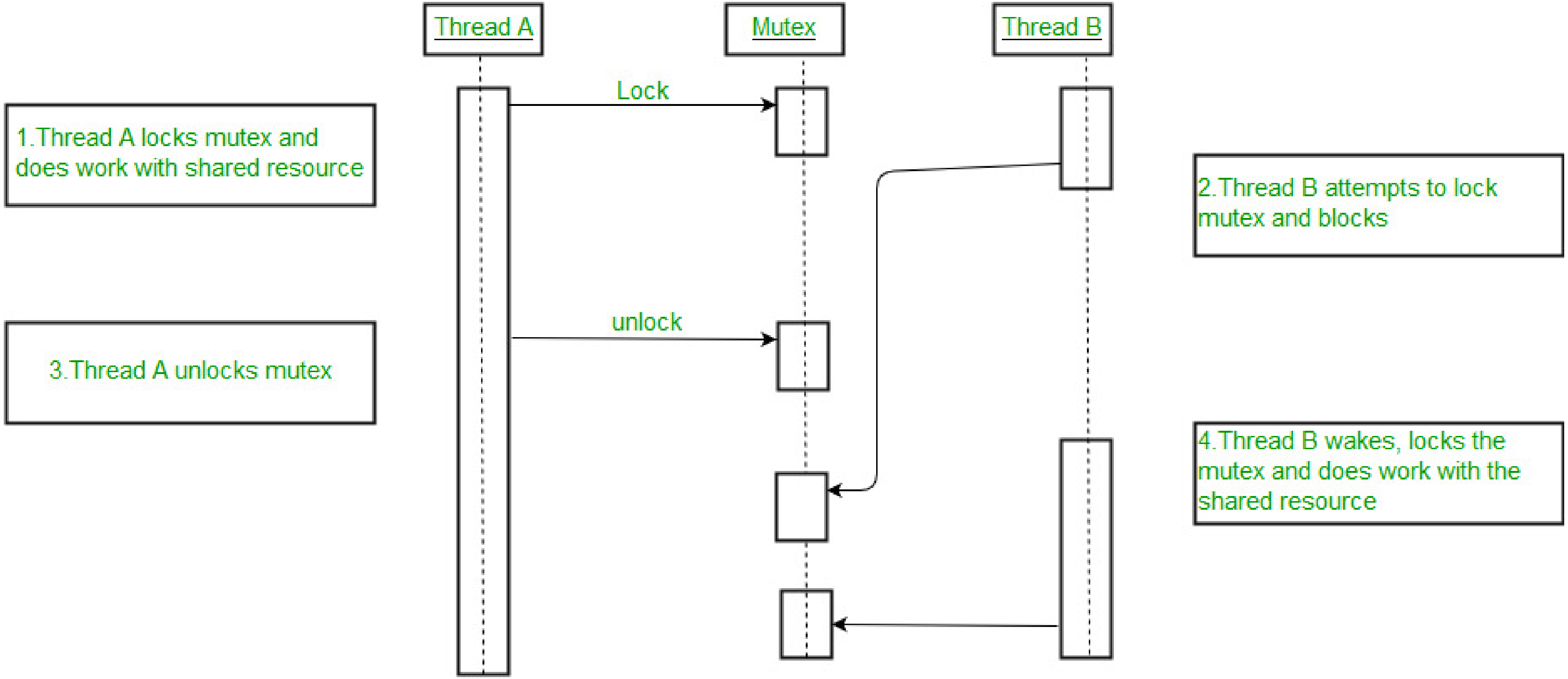
pthread Syntax: Mutex

- Syntax
  - `pthread_mutex_t mutex;` //In Global Section
  - `pthread_mutex_init(&mutex, NULL);` //Before create thread
  - `pthread_mutex_lock(&mutex);` //Entre CR
  - `pthread_mutex_unlock(&mutex);` //Exit CR
  - `pthread_mutex_destroy(&mutex);` //After all threads join
- Mutex
  - Wait till acquire the critical section.



- Examples
  - Can be used for any critical regions
  - **Simulate Bank Transaction:** Three persons are withdrawing 20, 2000 and 60 dollars respectively from a shared account at the same time. Again simultaneously a credit of 40,000 dollars has been made into the same account. Simulate these transactions using threads among which three will perform withdraw and one will perform credit. If the amount was 3,00,000 before any transaction, then find out the current amount by simulating the threads.

pthread Syntax: Mutex

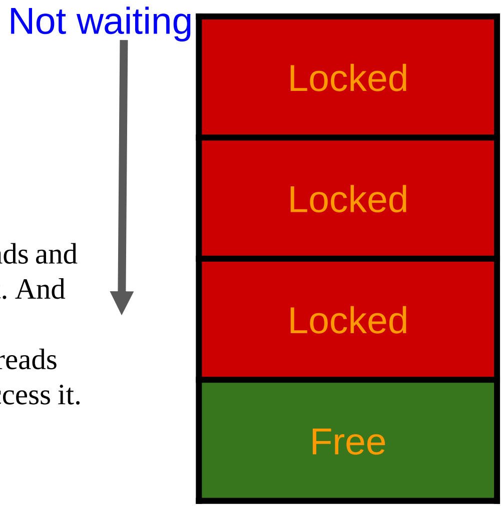


pthread Syntax: Try Mutex

- Syntax

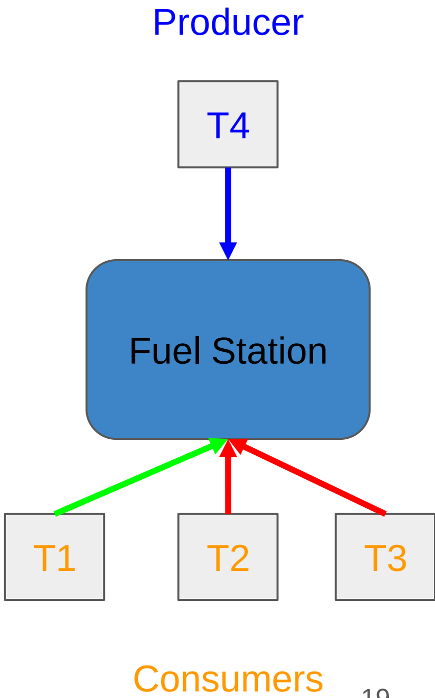
```
pthread_mutex_t mutex; //In Global Section
pthread_mutex_init(&mutex, NULL); //Before create thread
pthread_mutex_trylock(&mutex); //Returns 0 if successfully locked the resource
pthread_mutex_unlock(&mutex); //Exit CR
pthread_mutex_destroy(&mutex); //After all threads join
```

- Try Mutex
  - Does not wait to acquire the critical section
  - Multiple Mutexes and multiple resources
  - Each resources are using different locks
  - Use *try lock* to see which one is free and lock that
  - Don't wait in one resource
- Examples
  - **Simulate Kitchen room (Multiple stoves and multiple Chefs):** Where chefs represent threads and stoves represent the resources. A chef can use a stove at a single time, others can not access it. And find out how many times each stove has been used by simulating the threads.
  - **Simulate Washroom (Multiple Toilets and multiple Persons):** Where persons represent threads and toilets represent the resources. A person can use a toilet at a single time, others can not access it. And find out how many times each toilet has been used by simulating the threads.



pthread Syntax: Conditional Variable

- Syntax
  - `pthread_cond_t condVar;` //In Global Section
  - `pthread_cond_init(&condVar, NULL);` //Before create thread
  - `pthread_cond_wait(&condVar, &mutex);` **//Relinquish Mutex Lock**
    `pthread_cond_signal(&condFuel);` //Exit CR
  - `pthread_cond_destroy(&condVar);` //After all threads join
- Conditional Variable
  - To relinquish a lock
  - Give access to other different threads in the CR for a particular valid reason (condition)
  - Same mutex lock in two different threads
- Examples
  - Three Consumer threads are trying to access the Fuel Station
  - T1 gets the lock and T2, T3 are waiting for the lock
  - But T1 sees there is not enough fuel here
  - So he called the authority and waiting for the signal
  - Then the authority (T4) locks the Fuel and adds some fuel to it
  - Then T4 provides signal to T1 thread



pthread Syntax: Thread Synchronization

- Barrier
  - A barrier for a group of threads or processes in the source code means any thread/process must stop at this point and cannot proceed until all other threads/processes reach this barrier.

- Semaphore
  - Semaphore is **an integer variable which is used as a signal** to allow or not allow a process to access the critical section of the code or certain other resources. There are two types of semaphores:
    - **Binary Semaphore -** take on values 0 or 1.
    - **Counting Semaphore -** take on any integer value.

pthread Syntax: Barrier

- Syntax
  - `pthread_barrier_t barrier;`
  - `pthread_barrier_init(&barrier, NULL, 10);`
  - **`pthread_barrier_wait(&barrier)`**`;`

    -- The pthread_barrier_wait() function shall synchronize participating threads at the barrier referenced by barrier.
  - `pthread_barrier_destroy(&barrier);`
- Barrier
  - In barrier point all threads need to be present (all threads Program Counter or Instruction Pointer must be same) and then start execution independently.

The init function :

```
pthread_barrier_init(&our_barrier,NULL,4);
```

initializes the barrier "our_barrier" to wait for four threads.

Barriers

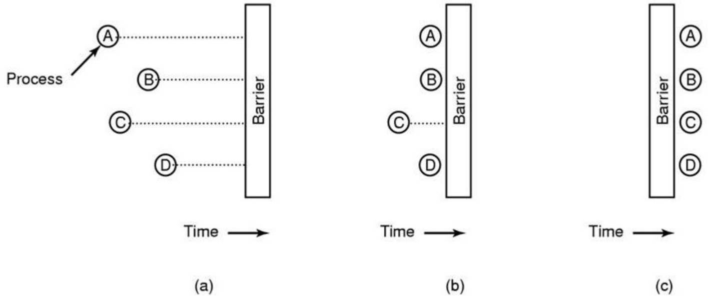


Figure 2-37. Use of a barrier. (a) Processes approaching a barrier. (b) All processes but one blocked at the barrier. (c) When the last process arrives at the barrier, all of them are let through.

pthread Syntax: Semaphore

- Syntax
  - `sem_t semaphore;`
  - `sem_init(&semaphore, 0, #threads);`

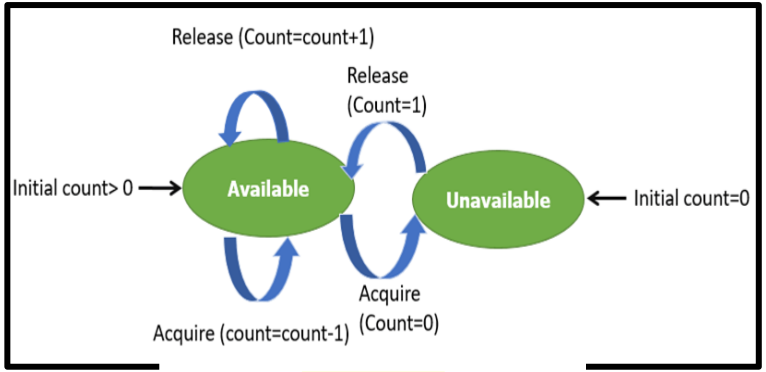
  (The sem_init() function is used to initialise the unnamed semaphore referred to by sem.)
  - `sem_wait(&semaphore);`

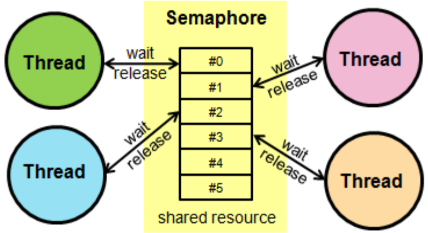
  The sem_wait() function decrements by one the value of the semaphore.
  - `sem_post(&semaphore);`

  (The sem_post() function unlocks the semaphore referenced by sem by performing a semaphore unlock operation on that semaphore)
  - `sem_destroy(&semaphore);`

  The sem_destroy() function destroys an unnamed semaphore that was previously initialized.

- Semaphore
  - Multiple resources; It allows a number of threads to access shared resources.
  - Multiple thread can access the same critical region





Examples

Calculate the sum of an array of size *n* using pthread. Where there is *m* threads and the i^{th} thread locally calculates the sum from *(n/m*i)* to *(n/m*(i+1) - 1).*

Value	31	42	27	59	1	20	120	77	12	3
Index	0	1	2	3	4	5	6	7	8	9

Say the array size is **10(n)**. And there are **2(m) threads**.

Then 1st thread will calculate:
sum of arr[0] to arr[4] = 31+42+27+59+1 =
And 2nd thread will calculate:
sum of arr[5] to arr[9] = 20+120+77+12+3 =

THANK YOU