Advanced

- Thread Management
- Pausing & Resuming Threads
- Thread Pools/Work Crew Model
- Thread Barriers
- Wait-Queue Model
- Event Pair Synchronization
- Thread Monitor
    - Solving Reader Writer Problem
    - Solving Bridge Problem
    - Strict Alternation using Monitor
    - Producer-Consumer Problem
- Assembly Line Scheduling
- Building Timers using Threads*
- Designing Multi-Threaded TCP Server
- Deadlock Detection and Prevention
- FiFo Semaphores
- Inter-Process Synchronization using Semaphores
- Forking a Multi-Threaded Program

Advanced Multithreading in C/C++ (Pthreads)

Pre-Requisite :

Must have Completed Prequel Course

Or

Have Good knowledge of Mutexes & CVs

Thread Synchronization Advanced → Thread Management

- In order to implement advanced concepts based on multi-threading, it is required that developer has to have a full control over an execution Unit – a thread

- Until now, all we had was thread_handle pthread_t using which developer could perform certain operations on a thread
  - Cancelling the thread
  - Joining the thread

- In order to implement advanced concepts based on multi-threading, Developer need to track the complete status of each thread
  - Is the thread executing or blocked ?
  - Is the thread blocked on CV, then on which CV ?
  - Is thread a reader thread or writer thread ?
  - What are the resources already allocated to the thread ?
  - Is the thread Queued up in the thread pool ?
  - etc

- We can't achieve tracking the complete status of the track in our application using just a thread_handle, we need a data structure with several members which represents a thread

- Using this wrapper thread data structure, we shall be able to manage our threads, Implement Adv thread Sync Data structures easily and in a much manageable way

Thread Synchronization Advanced → Thread Management

```
typedef struct thread_{

    char name[32];
    bool thread_created;
    pthread_t thread;
    void *arg;
    void *(*thread_fn)(void *);
    pthread_attr_t attributes;
} thread_t;
```

APIs :

```
thread_t *
thread_create (thread_t *thread, char *name);

void
thread_set_thread_attribute_joinable_or_detached(
                thread_t *thread, bool joinable) ;

void
thread_run (thread_t *thread, void *(*thread_fn)(void *), void *arg);
```

Reference : MultithreadingBible/ThreadSyncAdv/threadlib/threadlib.h
MultithreadingBible/ThreadSyncAdv/threadlib/threadlib.c

Course that we will build in this course :

MultithreadingBible/ThreadSyncAdv/threadlib/CourseDev/section_threadmgmt/threadlib.h

MultithreadingBible/ThreadSyncAdv/threadlib/CourseDev/section_threadmgmt/threadlib.c

Pausing

&

Resuming

Threads

Dir : MultithreadingBible/ThreadSyncAdv/threadlib/CourseDev/section_thread_pause

- Pausing and Resuming a Thread in the middle of its execution is a common scenario in the Multi-threading World
- Pausing and Resuming Threads :
  - A GC thread need to be paused, because application has triggered some high priority thread
  - A thread sending network packets into the network may want to be paused if network is congested
- We Can't pause a thread just at any random instruction in its execution flow, it will lead to the problem of invariants
  (Same reasoning as cancelling the thread, revise)
- Just like we have Cancellation points, we need to have Pause points – Instructions where if thread is paused then problem of invariants won't occur
- Cancellation points and Pause points usually overlap
- Identifying the pause points as per the same logic as cancellation points
- POSIX standard, like Thread Cancellation, do not provide inbuilt support for Pausing and Resuming threads, it is something we need to built ourself using Mutex and CVs

- Developer must be able to mark the thread to get paused

```
void
thread_pause(thread_t *thread);
```

- Once the thread is marked to pause, thread must get pause at the pause-points

```
void
thread_test_and_pause(thread_t *thread);
```

- Developer should be able to resume the paused thread

```
void
thread_resume(thread_t *thread);
```

- When thread wakes up (resumes), thread must invoke an additional registered fn immediately on wake up before it actually continues with work where it had left before pause

```
void
thread_set_pause_fn(thread_t *thread,
        void *(*thread_pause_fn)(void *),
        void *pause_arg);
```

```
typedef struct thread_{
        . . .
        . . .

        /* Fn to be invoked just before pausing the thread */
    void *(*thread_pause_fn)(void *);
    /* Arg to be supplied to pause fn */
    void *pause_arg;
    /* track thread state */
    uint32_t flags;
    /* update thread state mutually exclusively */
    pthread_mutex_t state_mutex;
    /* cv on which thread will block itself*/
    pthread_cond_t cv;

    . . .
    . . .
} thread_t;
```

- Update thread_create( )

```
typedef struct thread_{
        . . .
        uint32_t flags;
        . . .
} thread_t;
```

```
/* When the thread is running and doing its work as normal */
#define THREAD_F_RUNNING              (1 << 0)

/* When the thread has been marked to pause, but not paused yet */
#define THREAD_F_MARKED_FOR_PAUSE     (1 << 1)

/* When thread is blocked (paused) */
#define THREAD_F_PAUSED               (1 << 2)

/* When thread is blocked on CV for reason other than paused */
#define THREAD_F_BLOCKED              (1 << 3)
```

Thread Pools

(Park your Threads in a Garage !)

Imagine you are super rich, and there are couple of servants always in ready state waiting for your next commands to do your work.
Life will be going to be easy, no !

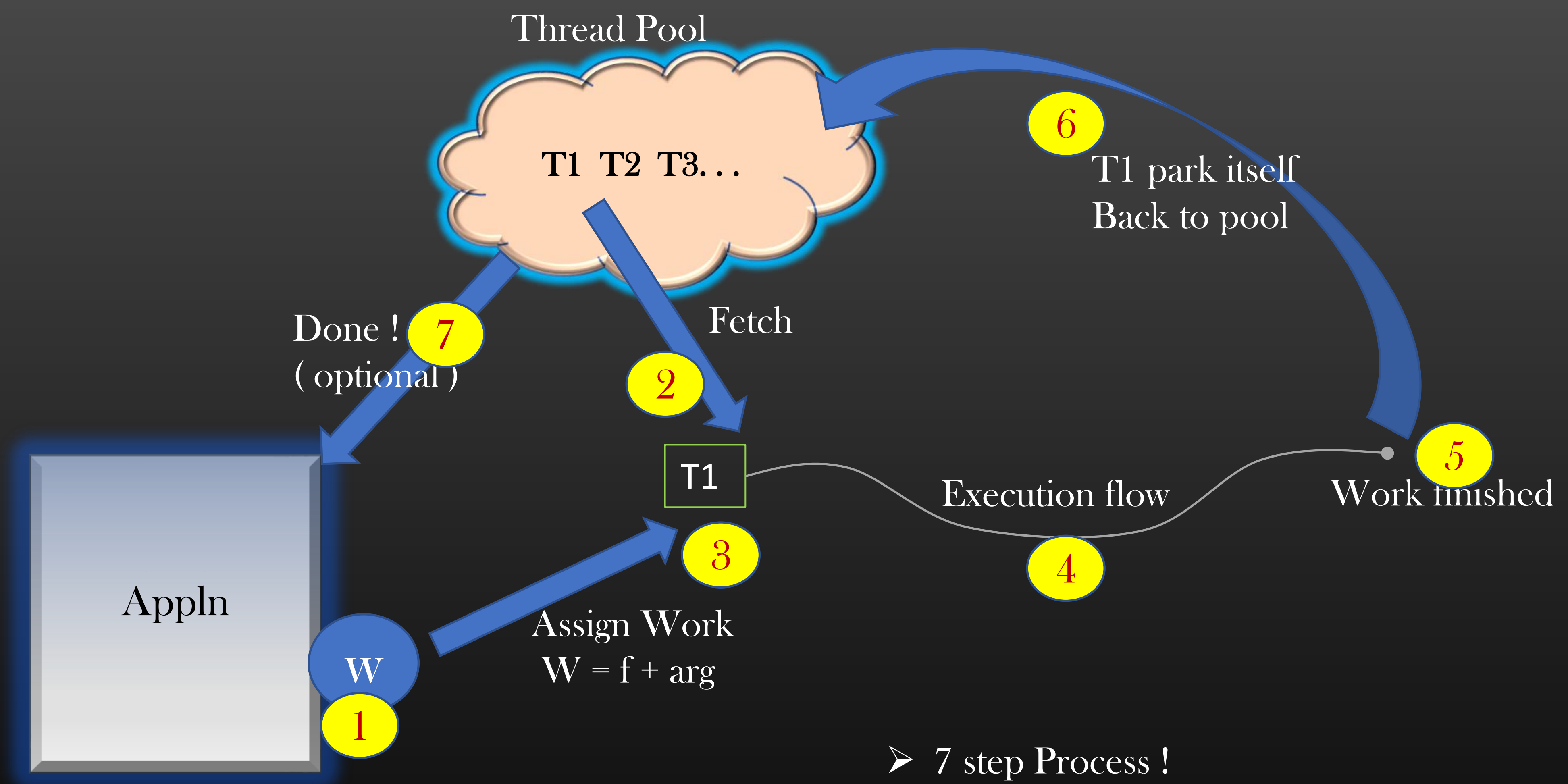
Dir : MultithreadingBible/ThreadSyncAdv/threadlib/CourseDev/section_thread_pools

- A Thread Pool is a parking space for blocked threads (but ready to resume)
  - Thread pool can be modeled as any data structure (list, tree, etc) which can hold thread_t objects
  - A thread in a thread pool is in ready-to-use state, just like you have your cars parked in your garages are in ready-to-use state
  - Threads blocked on CVs are placed (stored) in thread pools for later use
  - When appln needs to do a work W, it picks up an unused thread from threads pool, assign W to the thread and signal the thread
  - A thread is placed back in thread pool after it has completed its work
  - An application, in init phase, can create some pre-defined number of threads in thread pool
  - This software pattern is also called as Worker-Crew Pattern

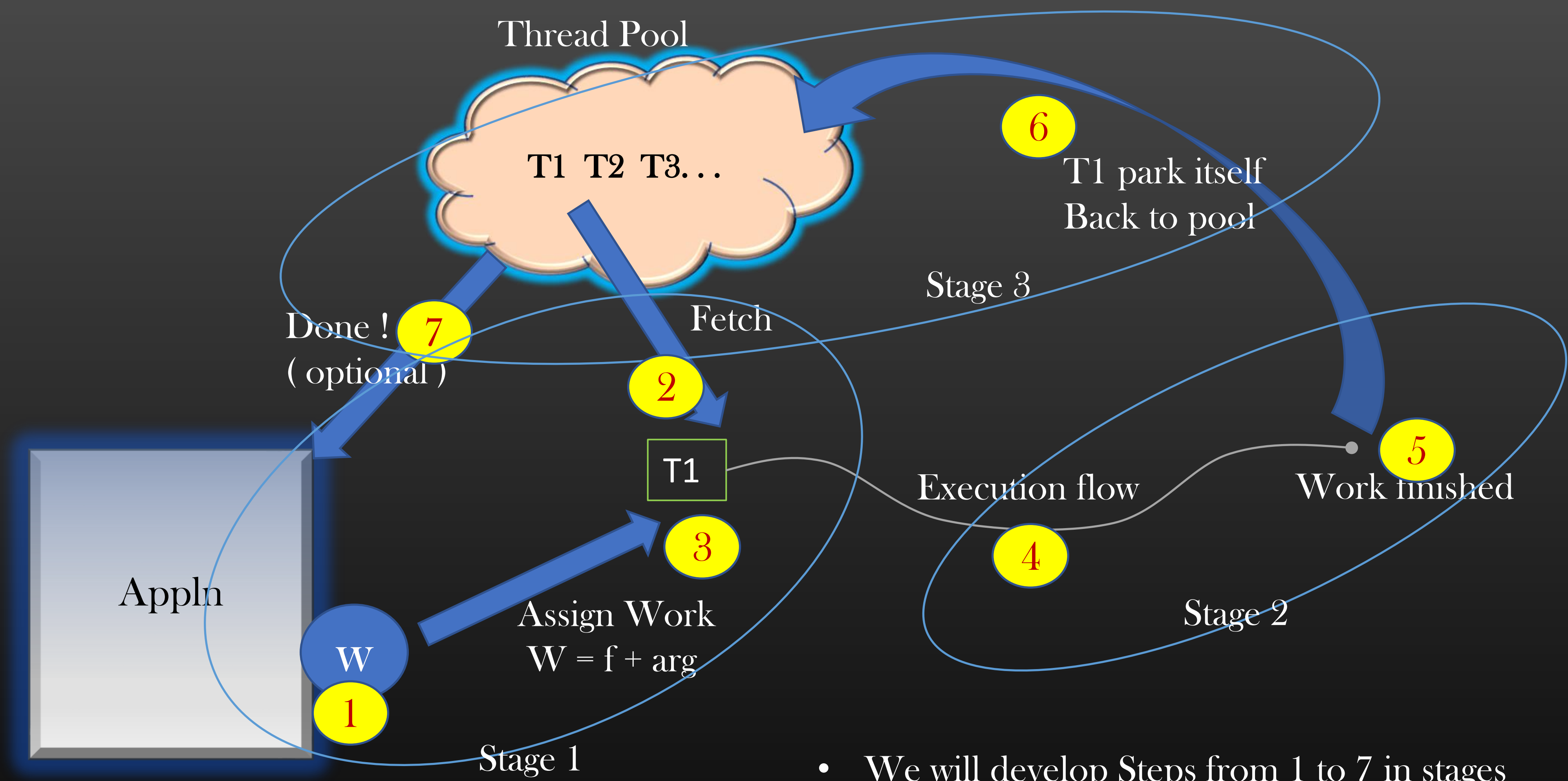
  - In this Section, we shall be going to implement a thread-pool functionality to our threadlib library

- Benefit
  - Application don't have to call pthread_create( ) to create new thread (performance improvement)
  - pthread_create( ) is an expensive call as it is a wrapper over clone( ) system call
  - Eg : Network Appln processing incoming pkts in the context of a separate thread
  - Places an upper bound on number of threads an application can create. Useful in resource constraint environment
  - Eg : A thread which has established connection with remote host can be re-used again & again from thread pool without dropping and re-establishing a connection again

    Read Wiki :
    https://en.wikipedia.org/wiki/Thread_pool



- 7 step Process !
- Step 1 , 2 and 3 is performed by the application
- Step 4, 5, 6 and 7 is performed by the worker thread



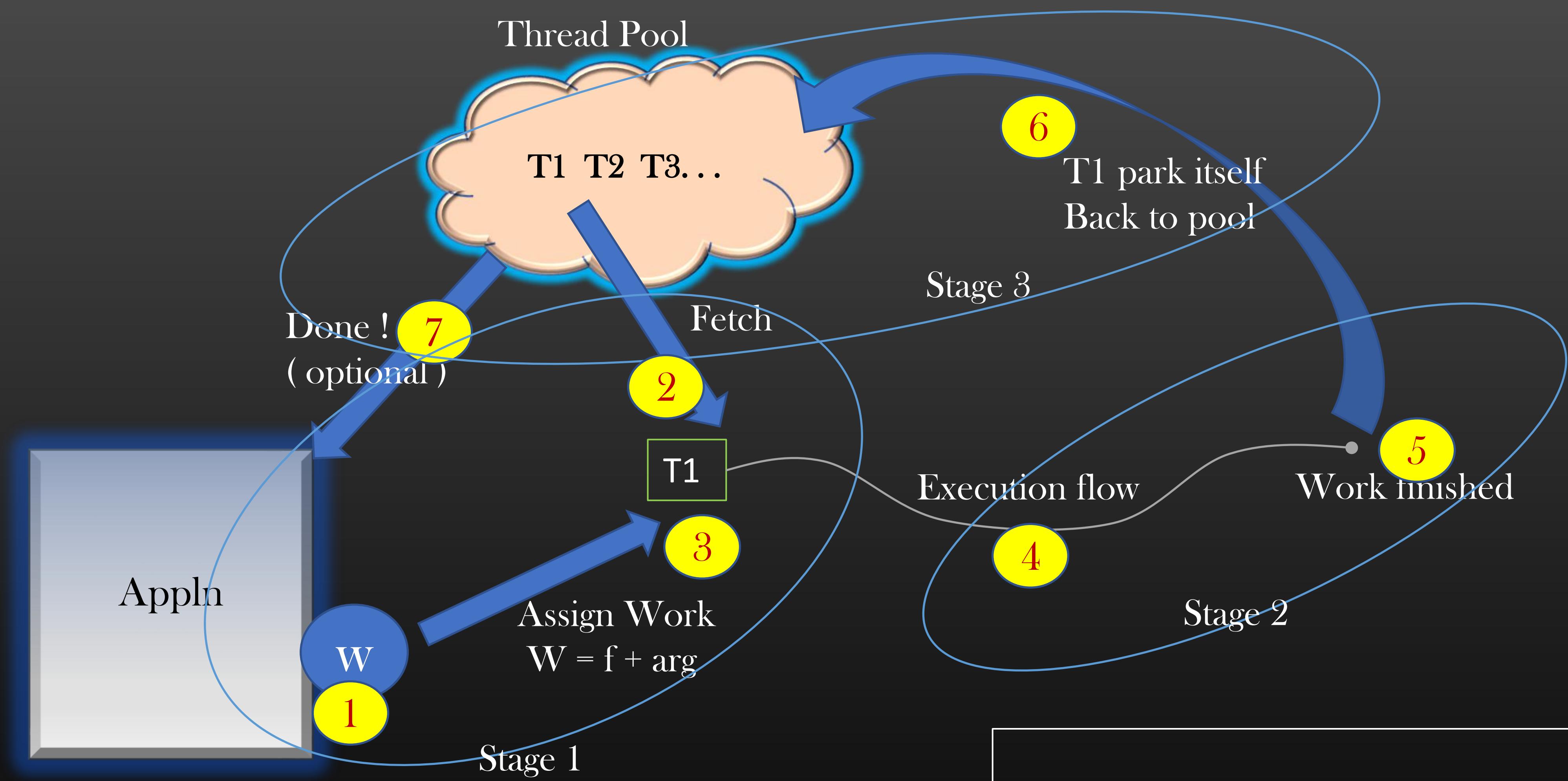
- We will develop Steps from 1 to 7 in stages
- Steps 1 , 2 & 3 – Stage 1
- Step 4 & 5 – Stage 2
- Step 6 & 7 – Stage 3
- Each stage is a separate fn

```
typedef struct thread_pool_ {

  glthread_t pool_head;
  pthread_mutex_t mutex;
} thread_pool_t;


void
thread_pool_init (thread_pool_t *th_pool );

void
thread_pool_insert_new_thread (thread_pool_t *th_pool, thread_t *thread);

thread_t *
thread_pool_get_thread (thread_pool_t *th_pool);

void
thread_pool_dispatch_thread (thread_pool_t *th_pool,
                             void *(*thread_fn)(void *),
                             void *arg);
```

```
typedef struct thread_{

    . . .
    glthread_t wait_glue;
               . . .
} thread_t;
GLTHREAD_TO_STRUCT(wait_glue_to_thread,
       thread_t, wait_glue);
```

Helper APIs :

```
/*  Part the thread back in thread pool
   Algo :  1. Add the thread_t object to thread pool
                  2. Block on CV
*/
static void
thread_pool_thread_stage3_fn (thread_pool_t *th_pool,
        thread_t *thread) ;

/*  Run the thread
     Assume : Thread has been removed from thread pool already
     Algo :  1. If execution unit is not created yet, create it
                        2. else signal the thread (bcoz thread is blocked on CV )
*/
static void
thread_pool_run_thread (thread_t *thread);
```



- Try yourself !
- Step 1 to 6 (7 later)

```
void
thread_pool_dispatch_thread (thread_pool_t *th_pool,
                             void *(*thread_fn)(void *),
                             void *arg);
```

- Stage 2 is all about executing appln's thread_fn on appln's arg (performing appln's given task)
- We need to figure out a way how thread can know that it has to execute stage3 fn after stage 2
- There should be some data structure which stores the info regarding how thread should execute through stages !

```
void
thread_pool_dispatch_thread (thread_pool_t *th_pool,
                             void *(*thread_fn)(void *),
                             void *arg);
```

```
typedef struct thread_execution_data_ {

    void *(*thread_stage2_fn)(void *);

    void *stage2_arg;

    void (*thread_stage3_fn)(thread_pool_t *, thread_t *);

    thread_pool_t *thread_pool;

    thread_t *thread;

} thread_execution_data_t;
```

Thread Synchronization → Thread Pools → Data Structure and APIs

```
static void *
thread_fn_execute_stage2_and_stage3 (void *arg)  {

    thread_execution_data_t *thread_execution_data =
            (thread_execution_data_t *)arg;

     while ( 1 ) {
       /* Stage 2 : User defined function with user defined argument*/
        thread_execution_data->thread_stage2_fn (thread_execution_data->stage2_arg);
        /*  Stage 3 : Queue the thread in thread pool and block it*/
        thread_execution_data->thread_stage3_fn (thread_execution_data->thread_pool,
                                          thread_execution_data->thread);
   }
}
```

```
typedef struct thread_execution_data_ {

    void *(*thread_stage2_fn)(void *);

    void *stage2_arg;

    void (*thread_stage3_fn)(thread_pool_t *, thread_t *);

    thread_pool_t *thread_pool;

    thread_t *thread;

} thread_execution_data_t;
```

- For every use of thread from thread-pool, appln needs to change the stage 2 fn and arg before signaling the blocked thread

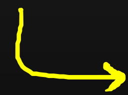
- Step 7

Thread Pool
T1 T2 T3. . .
6
T1 park itself
Back to pool
Stage 3
Done ! 7
(optional)
Fetch
2
T1
Execution flow
5
Work finished
Appln
W
1
3
Assign Work
W = f + arg
4
Stage 2
Stage 1

```
void
thread_pool_dispatch_thread (thread_pool_t *th_pool,
                             void *(*thread_fn)(void *),
                             void *arg, bool block_caller);
```

- Application should block until the worker thread completes its task and park itself in thread pool
- Hint : Use Zero Semaphore

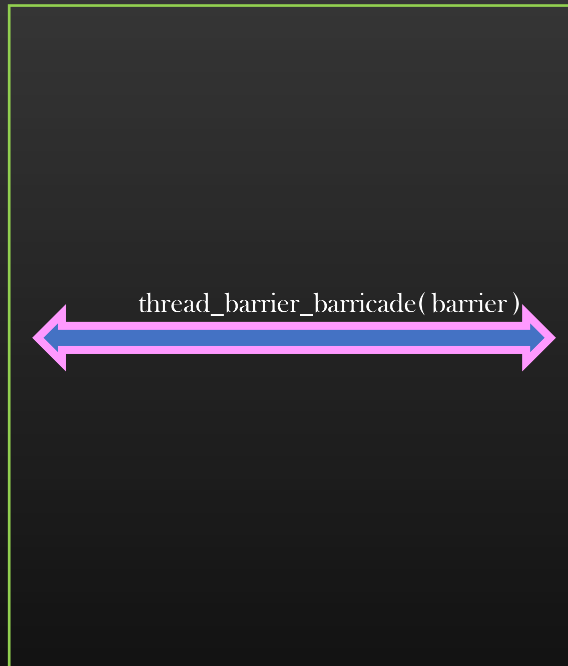
Thread Barrier



- You are starting a trip with your friends; you wait at some point to enable rest of your friends to join you

- Thread Barrier is a thread synchronization data structure which blocks all threads at a particular line of code until some specified # of threads arrives at the barrier point

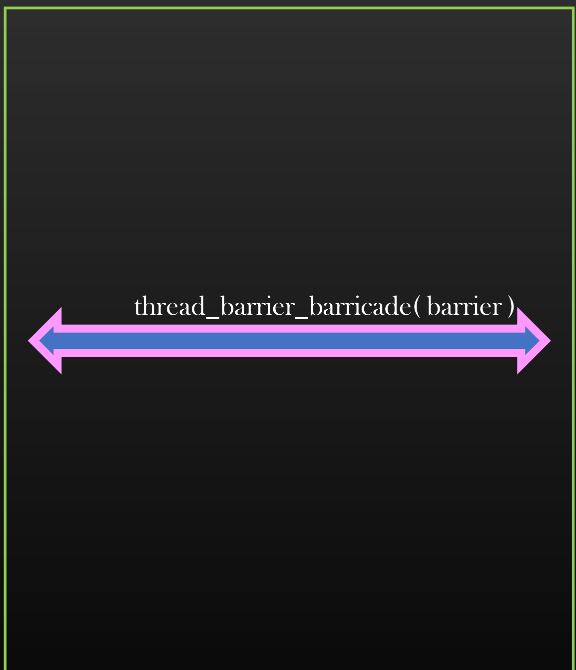
- In cases where you must wait for a number of tasks to be completed before an overall task can proceed, barrier synchronization can be used



POSIX API provide in built support to work with Thread barriers (pthread_barrier_t),
but it is no fun if we don't know how to implement one from scratch !

- Practical use case : IDM appln downloads the big files through multiple downloader threads. When all threads complete, then only application reports file download success

- Thread Barrier works on the concept of relay
  - A signaled thread, signaling the other blocked thread to resume – creating a chain of signals
  - When we get stuck at barricade, this is how we pass the barricade when allowed by authority – one by one ! No ?

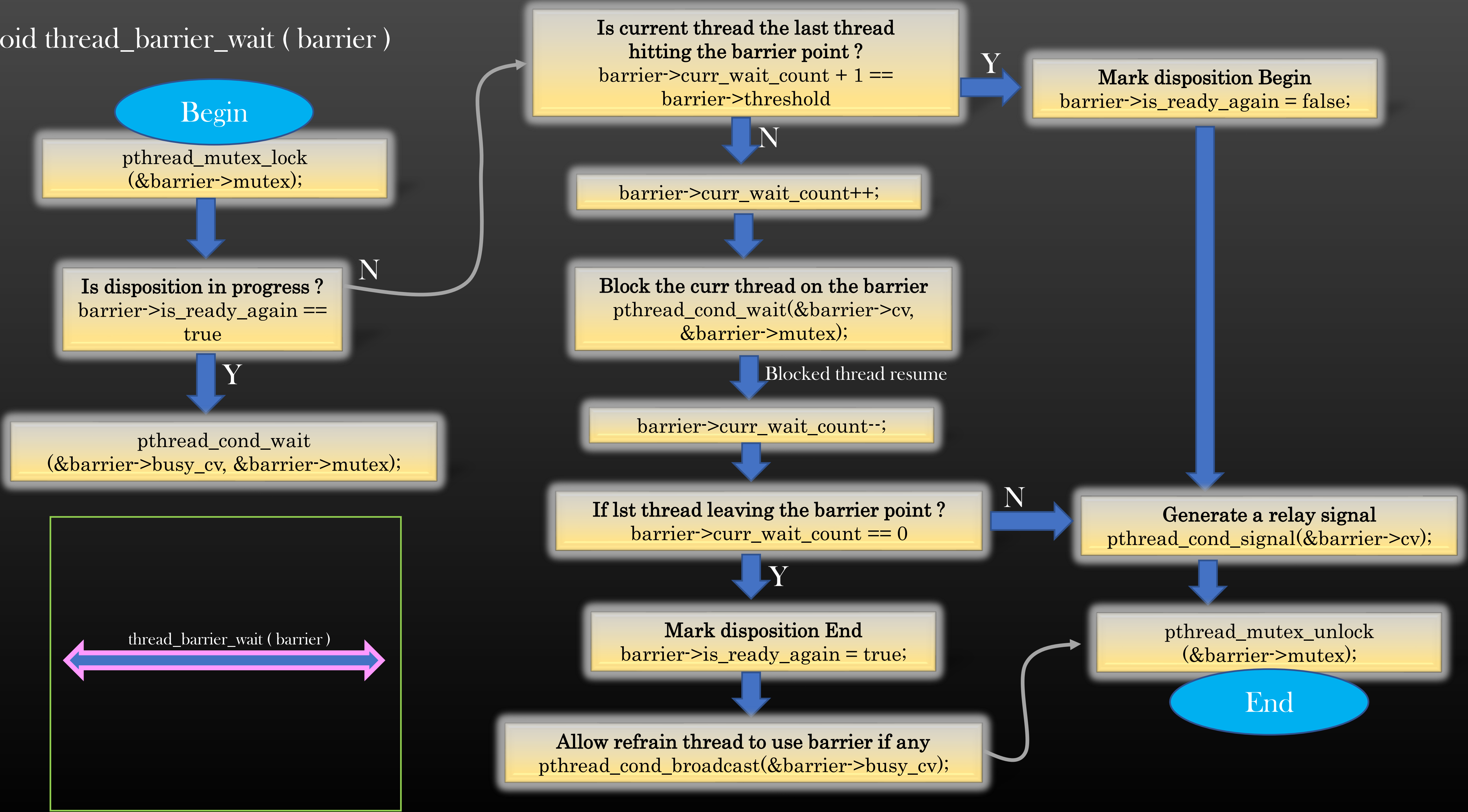



```
typedef struct th_barrier_ {

        uint32_t max_count;
        uint32_t curr_wait_count;
        pthread_cond_t cv;
        pthread_mutex_t mutex;
        bool is_ready_again;
        pthread_cond_t busy_cv;
} th_barrier_t
```

```
void
thread_barrier_init ( th_barrier_t *barrier, uint32_t threshold);
```

```
void
thread_barrier_init ( th_barrier_t *barrier, uint32_t threshold_count);

void
thread_barrier_wait ( th_barrier_t *barrier);

void
thread_barrier_destroy ( th_barrier_t *barrier );
```

https://github.com/sachinites/MultithreadingBible
Codes : MultithreadingBible/ThreadBarrier



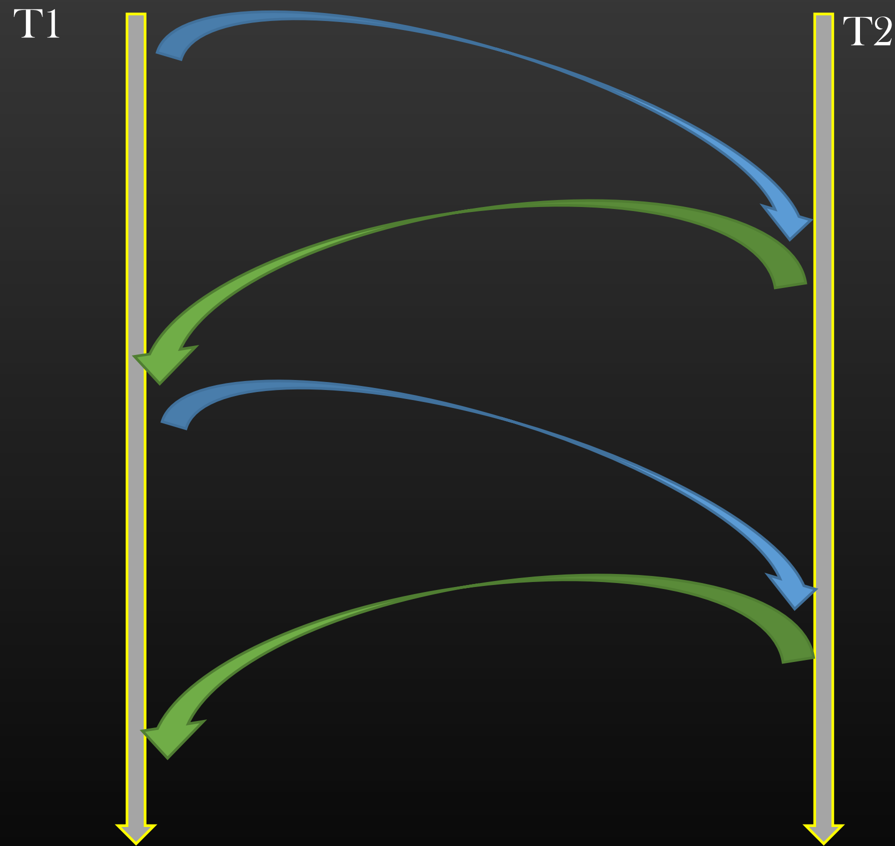
void thread_barrier_wait (barrier)
Begin
pthread_mutex_lock (&barrier->mutex);
Is disposition in progress ? barrier->is_ready_again == true
N
Y
pthread_cond_wait (&barrier->busy_cv, &barrier->mutex);
thread_barrier_wait (barrier)
Is current thread the last thread hitting the barrier point ? barrier->curr_wait_count + 1 == barrier->threshold
Y
N
Mark disposition Begin barrier->is_ready_again = false;
barrier->curr_wait_count++;
Block the curr thread on the barrier pthread_cond_wait(&barrier->cv, &barrier->mutex);
Blocked thread resume
barrier->curr_wait_count--;
If lst thread leaving the barrier point ? barrier->curr_wait_count == 0
N
Y
Generate a relay signal pthread_cond_signal(&barrier->cv);
Mark disposition End barrier->is_ready_again = true;
pthread_mutex_unlock (&barrier->mutex);
End
Allow refrain thread to use barrier if any pthread_cond_broadcast(&barrier->busy_cv);

Event Pair

(Strict Alternation using Semaphore)

Thread Synchronization → Event Pair

- As the name suggest, Event Pair Data structure is used to implement strict alternation between a pair of threads
- We have already implemented strict alternation using semaphores, we would re-implement it again by wrapping the same concept under formal event pair APIs



- T1 waits after sending a msg
- T1 can send next msg only after receiving the reply of prev msg
- T2 waits after sending the msg
- T2 can send next msg only after receiving the reply of prev msg

Wait Queues

Thread Synchronization → Wait Queues → Functioning

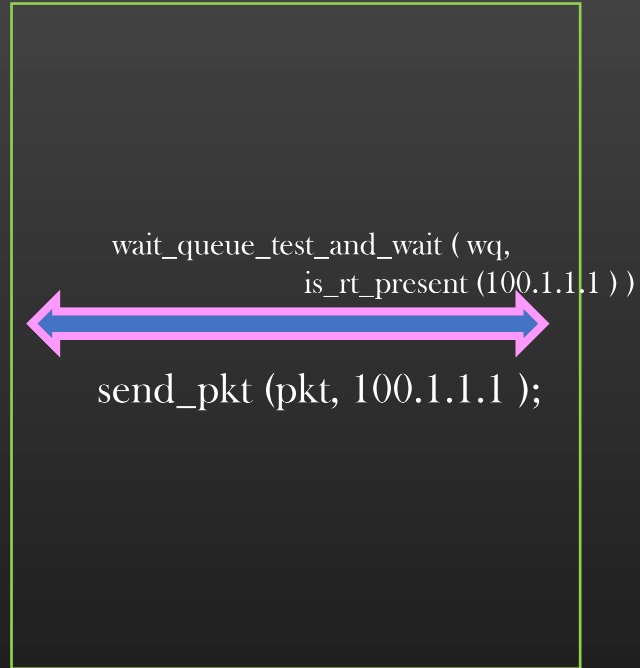
- Wait Queue (WQ) , as the name suggest, is a thread synchronization data structure
- WQs holds the thread(s) and keep them in blocked state until some specified condition is met
- Used extensively in Linux kernel, we shall be implementing wait-Queues for user space programs
- POSIX thread library do not provide inbuilt WQs, we need to build one ourselves from scratch for our convenience

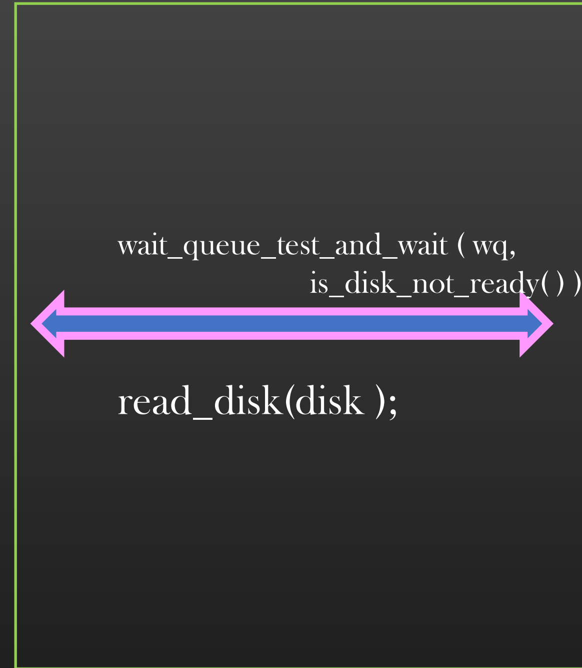
Eg : Thread starts sending the pkts to 100.1.1.1 as soon as route to 100.1.1.1 becomes available in rt table, until then it stays blocked

Eg : Thread(s) are waiting to read the disk until disk becomes ready to read

- We can implement the above logic with CV and Mutexes alone, without using WQs but then developer has to deal with Mutex and CVs
- WQs simplifies , developer don't have to write any thread synch code explicitly in his appln
- Wait Queues hide all the thread synchronization complexity , abstracting away the users from dealing with mutexes & CVs
- Same is true with thread barriers if you had noticed

Thread Synchronization → Wait Queues → Usage





- As many different Application specific conditions, as many wait Queues to be defined
- Nothing new, just doing the same thing in slightly another way to implement WQs

Consumer Thread :
General pseudo to block on CV

```
pthread_mutex_lock(&mutex);

while ( predicate( ) ) {

    pthread_cond_wait (&cv, &mutex);

}

execute_cs_on_wake_up ( );

pthread_mutex_unlock(&mutex);
```

Producer Thread :
General pseudo to signal on CV

```
pthread_mutex_lock(&mutex);

if  ( !predicate( ) ) {

    pthread_cond_signal (&cv );

}
pthread_mutex_unlock(&mutex);
```

- Almost all thread synchronization data structures boils down to producer consumer or variants problem !
- Nothing new, just doing the same thing in slightly another way to implement WQs

```
obj – some ptr to application object
. . .
. . .
wait_queue_test_and_wait ( WQ ,
                           condition_fn, (void *)obj ) ;
. . .
. . .
```

```
bool
condition_fn ( void *arg,
               pthread_mutex_t **out_mutex) ;
```

- If condition_fn returns true, the calling thread must block
- If condition_fn returns false, all blocked threads on a wait Queue are signaled
- Condition fn is invoked in two modes :
  - Lock mode
    - Second parameter *out_mutex* is not null
    - Lock the application mutex before checking the condition
    - return the locked mutex in second output parameter *out_mutex*
    - return bool – condition result
  - Unlock mode
    - Second parameter *out_mutex* is null
    - return bool – condition result

```
bool is_disk_drive_ready ( void *arg,
          pthread_mutex_t **out_mutex) {

          disk_t *disk = (disk_t *)arg;

          if (out_mutex) {
                    pthread_mutex_lock(&disk->mut
                    *out_mutex = &disk->mutex;
          }

          return disk->ready;
}
```

threadlib.h / threadlib.c

```
typedef struct wait_queue_ {

    /* No of threads waiting in a wait-queue*/
     uint32_t thread_wait_count;

     /* CV on which multiple threads in wait-queue are
          blocked */
     pthread_cond_t cv;

     /* Application owned Mutex cached by wait-queue */
     pthread_mutex_t *appln_mutex;

} wait_queue_t;
```

Condition fn " fn ptr " declaration :

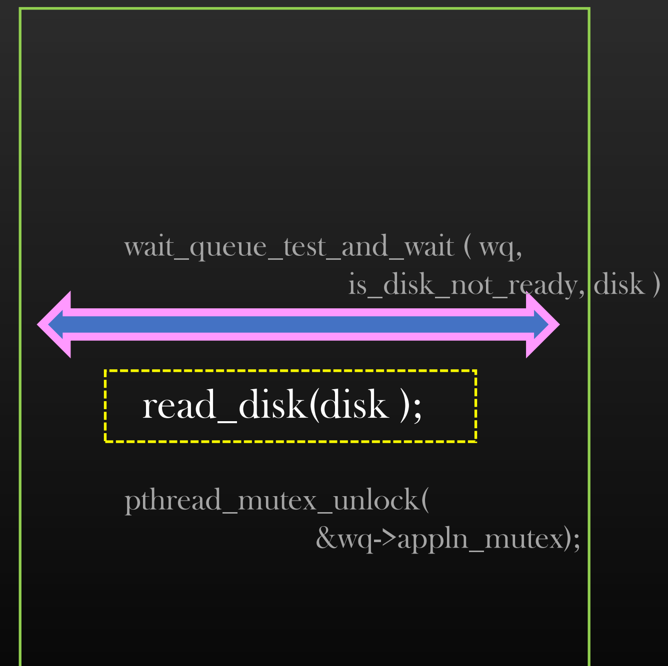
```
typedef bool (*wait_queue_condn_fn)
  (void *appln_arg, pthread_mutex_t **out_mutex);
```

```
void
wait_queue_init (wait_queue_t * wq);


thread_t *
wait_queue_test_and_wait (wait_queue_t *wq,
              wait_queue_block_fn wait_queue_block_fn_cb,
              void *arg );


void
wait_queue_signal (wait_queue_t *wq, bool lock_mutex);


void
wait_queue_broadcast (wait_queue_t *wq, bool lock_mutex);


void
wait_queue_destroy (wait_queue_t *wq);
```

Thread Synchronization → Wait Queues → test_and_wait( )

```
void
wait_queue_test_and_wait (
            wait_queue_t * wq,
            wait_queue_block_fn wait_queue_block_fn_cb,
            void *arg)
{
  bool should_block;
  pthread_mutex_t *locked_appln_mutex = NULL;

 should_block = wait_queue_block_fn_cb (arg,

            &locked_appln_mutex);

 wq->appln_mutex = locked_appln_mutex;

  while (should_block)
   {
     wq->thread_wait_count++;
     pthread_cond_wait (&wq->cv, wq->appln_mutex);
     wq->thread_wait_count--;
     should_block = wait_queue_block_fn_cb (arg, NULL);
   }
}
```

```
void
wait_queue_test_and_wait (
            wait_queue_t * wq,
            wait_queue_block_fn wait_queue_block_fn_cb
            void *arg) ;
```



```
void
wait_queue_test_and_wait (
                wait_queue_t * wq,
                wait_queue_block_fn wait_queue_block_fn_cb,
                void *arg)
{
  bool should_block;
  pthread_mutex_t *locked_appln_mutex = NULL;

 should_block = wait_queue_block_fn_cb (arg,

                &locked_appln_mutex);

 wq->appln_mutex = locked_appln_mutex;

  while (should_block)
   {
     wq->thread_wait_count++;
     pthread_cond_wait (&wq->cv, wq->appln_mutex);
     wq->thread_wait_count--;
     should_block = wait_queue_block_fn_cb (arg, NULL);
   }
}
```

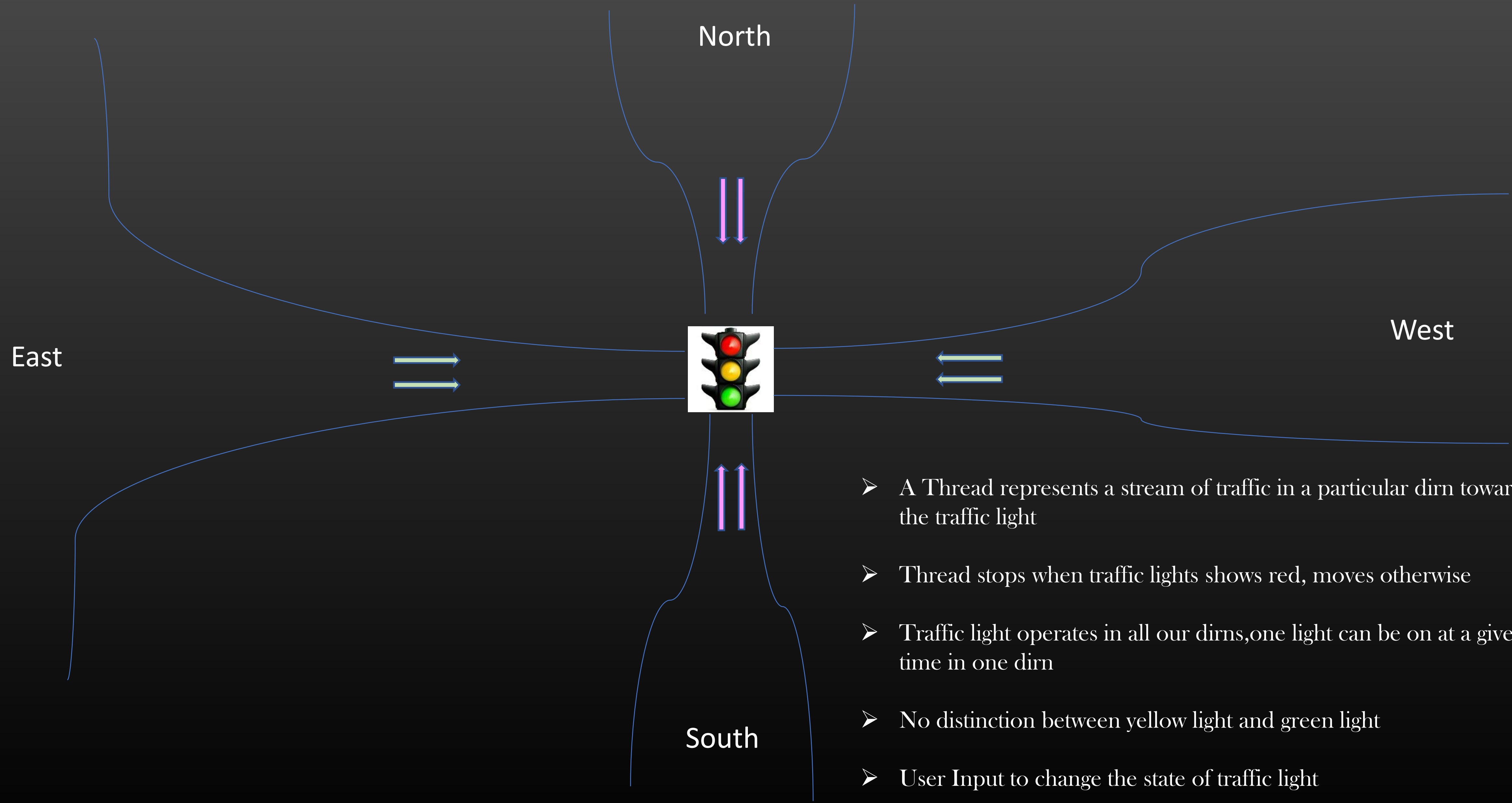
```
bool is_disk_drive_not_ready ( void *arg,
                pthread_mutex_t **out_mutex) {

                disk_t *disk = (disk_t *)arg;

                if (out_mutex) {
                        pthread_mutex_lock(&disk->mutex);
                        *out_mutex = &disk->mutex;
                }

                return disk->ready;
}
```

```
pthread_mutex_lock(&mutex);

while ( predicate( . . .) ) {

     pthread_cond_wait (&cv, &mutex);

}

execute_cs_on_wake_up ( );

pthread_mutex_unlock(&mutex);
```

General pseudo to block on CV
When Resource is busy

Let’s exercise Wait Queues

Thread Synchronization → Wait Queues → Demo Example → Problem Statement



North

Moving traffic as running threads

East

West

The moving threads must know :

1. Dirn in which they are moving
2. Their own state : moving or paused
3. The Traffic light Face

```
typedef struct thread_pvt_data_ {
    dirn_t th_dirn;
    th_state_t thread_state;
    thread_t *thread;
    tr_light_t *tr_light;
} thread_pvt_data_t;
```

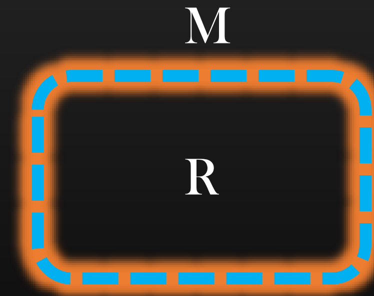
South

- Launch the threads to represents the Moving traffic
- Each face of traffic light will have its own WQ. On this WQ the threads (moving traffic) shall wait
- Let's code !
- tr_light.h / tr_light.c

Thread Monitors

Thread Synchronization → Thread Monitors → Introduction

- Thread Monitor is , yet another, and powerful thread synchronization data structure
- They are used to synchronize threads competing for an access on a shared resource
- Every resource has its personal dedicated monitor
- Monitors are analogues to traffic police man
  - Coordinate the threads competing for a resource
  - Monitor Algorithm is Cooperation Oriented
- Monitor is an interface between threads and a resource
  - Threads has to request monitor that it is interested in resource access
  - Threads announce to monitor that it is done with the resource and releasing it
  - Monitors are like protective case for a resource



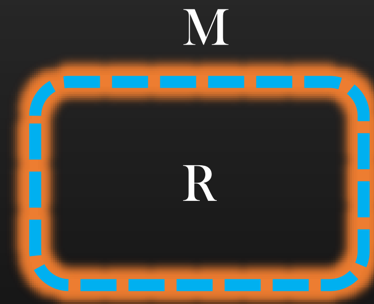
- Monitors keeps a track :
  - Whether the resource is being accessed by some thread or not
  - If accessed, then who is/are accessing it

- Read-Read is not conflicting operations
  - Monitors should allow multiple reader threads to access the resource

- Read-Write are conflicting operations
  - If reader threads(s) is/are accessing the resource then writer threads must wait
  - If Writer thread is accessing the resource then reader threads must wait

- Write-Write are conflicting operations
  - If Writer thread is accessing the resource then other writer threads must wait

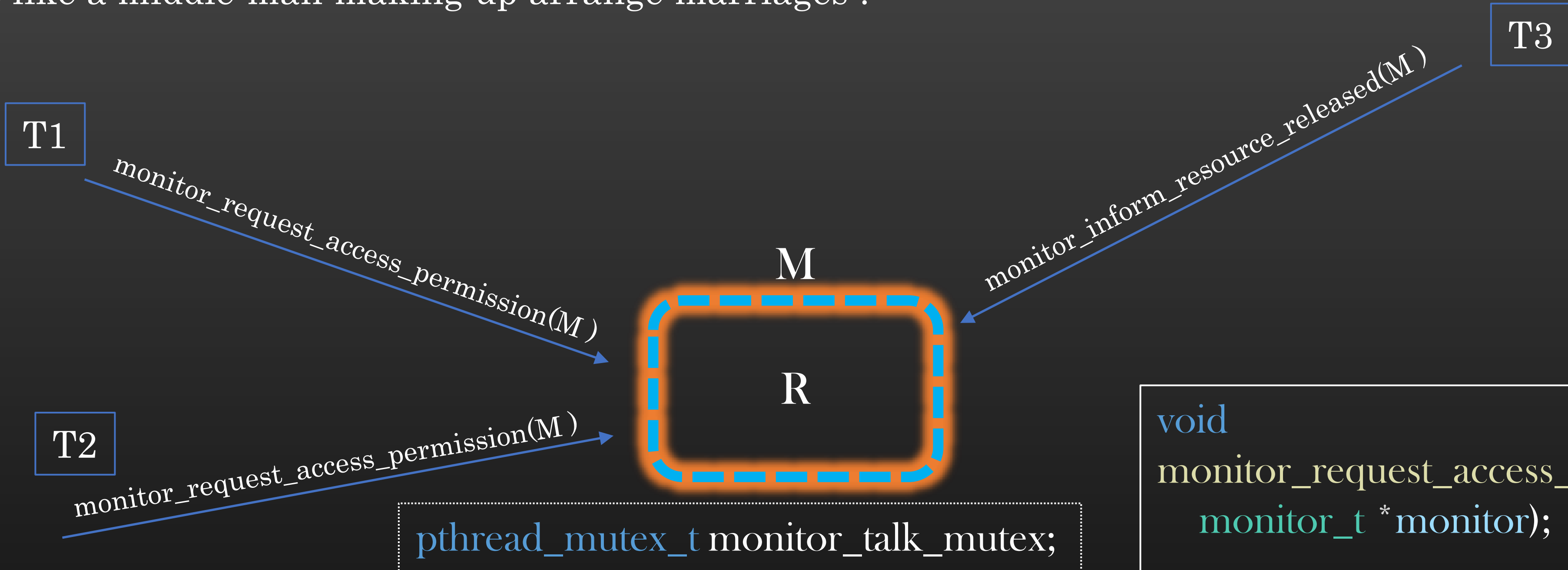
- If Resource is Idle, then any thread requesting the access to the resource must be granted an access



- Monitors may internally use Wait-Queues for its implementation

- Monitors can help us solve a large variety of complex thread synchronization problems with ease
  - Reader-Writer Problem
  - Bridge Problem

- Threads talk to Monitors to negotiate an access to a resource
- Threads do not coordinate amongst themselves (wait n signal)
- Monitor is like a middle-man making up arrange marriages !




```
void
monitor_request_access_permission (
    monitor_t *monitor);

void
monitor_inform_resource_released (
    monitor_t *monitor);
```

Multiple requester threads can issue request to monitor concurrently, to process the request (grant or not) monitor has to update its internal state in a mutually exclusive way. Therefore a monitor mutex is required.

- A developer always knows if he is creating a thread then what is the purpose of the thread
- So, threads can be classified as Reader thread or Writer thread
- Monitor keeps a track if the resource is being used by a reader thread or writer thread at any given point of time
- If a Resource is being accessed by a Reader thread, Monitor grants an access to next requester reader thread
- If a Resource is being accessed by a Writer thread, Monitor does not grant an access to next requester reader/writer thread and puts the requester thread in a wait Queue