1

High Performance Computing

Dr.Bheemappa H
IIIT, Sricity
bheemappa.h@iiit.in

M3 : Shared Memory Programming

- Thread Basics: Why Threads? The POSIX Thread API,
- Thread Creation and Termination,
- Synchronization Primitives in Pthreads,
- Controlling Thread and Synchronization Attributes,
- Thread Cancellation,
- Composite Synchronization Cons

Overview of Programming Models

- Explicit parallel programming
  - specification of parallel tasks along with their interactions.
    - synchronization between concurrent tasks or communication of intermediate results
- In shared address space architectures,
  - **communication** is implicitly specified since some (or all) of the memory is accessible to all the processors.
- Programming paradigms for shared address space machines focus on constructs for expressing concurrency and synchronization along with techniques for minimizing associated overheads

Preferred model for parallel programming

- Data sharing, concurrency models, and support for synchronization may vary based on **Shared address space programming paradigms.**

Process based models:

- Data associated with a process is private, by default, unless otherwise specified (protection in multiuser systems).
- Overheads associated with enforcing protection domains make processes less suitable for parallel programming.

Threads

lightweight processes and threads assume that all memory is global. By relaxing the protection domain, lightweight processes and threads support much faster manipulation. As a result, this is the preferred model for parallel programming

lightweight processes and threads are preferred model for parallel programming
"Directive based programming models"

Thread Basics

- A thread is a single stream of control in the flow of a program. A program like
  - All memory in the logical machine model of a thread is globally accessible to every thread

```
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
        c[row][column] =
            dot_product(get_row(a, row),
                        get_col(b, col));
```

can be transformed to:

```
for (row = 0; row < n; row++)
    for (column = 0; column < n; column++)
        c[row][column] =
            create_thread(
                dot_product(get_row(a, row),
                            get_col(b, col)));
```

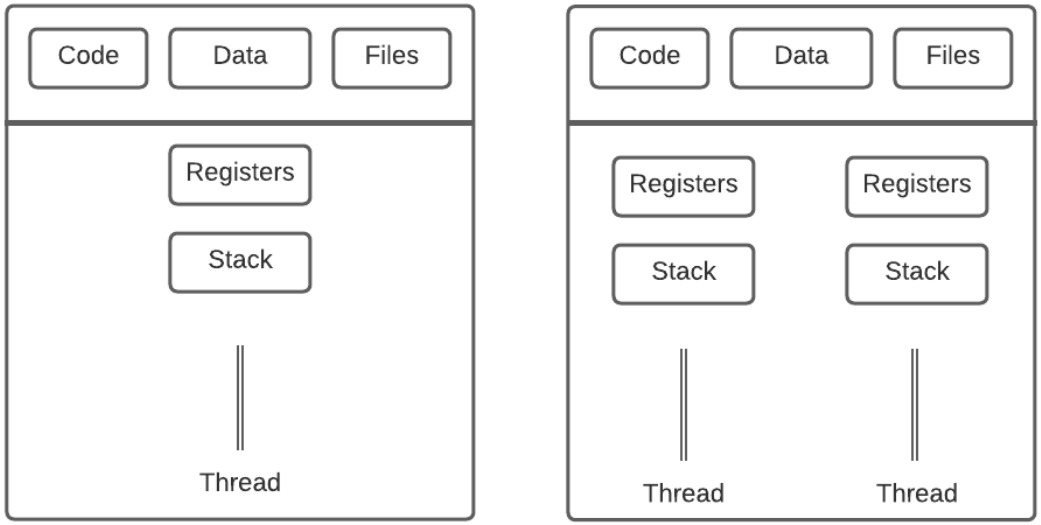
Why Threads?

- Threads provide software portability.
- Inherent support for latency hiding.
- Scheduling and load balancing.
- Ease of programming and widespread use

The POSIX Thread API

- Commonly referred to as **Pthreads**, POSIX has emerged as the standard threads API, supported by most vendor.
- The concepts discussed here are largely independent of the API and can be used for **programming with other thread APIs** (NT threads, Solaris threads, Java threads, etc.) as we

```
#include <pthread.h>
```



Single-threaded Process **Multi-threaded Process**

Thread Basics: Creation and Termination

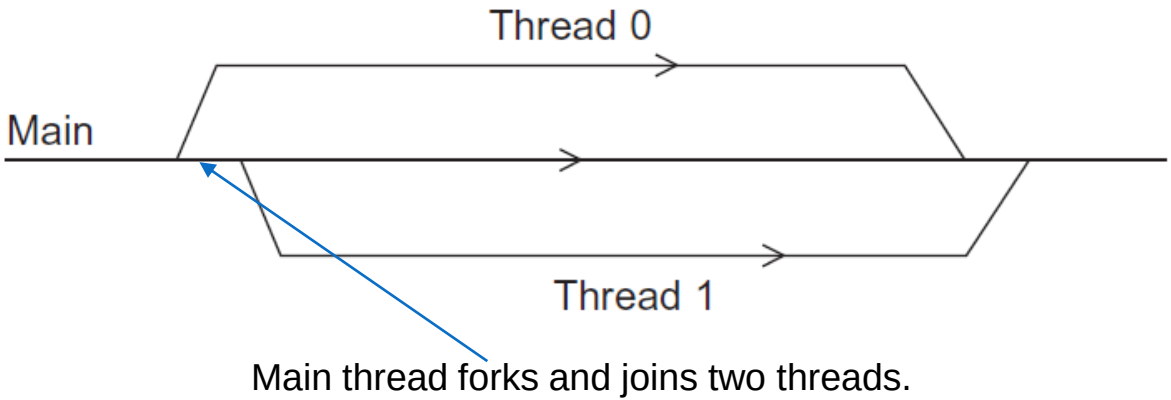
- pthread_create starts a new thread in the calling process
- pthread_create **returns zero on success and a nonzero error** number in the case of failure.
- pthread_join waits for the **thread identified by the first argument to terminate**.

```
#include <pthread.h>
int pthread_create (
    pthread_t   *thread_handle,
    const pthread_attr_t   *attribute,
    void *   (*thread_function)(void *),
    void   *arg);

int pthread_join (
    pthread_t   thread,
    void   **ptr);
```

Ex:01: creating a child thread

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void *worker_thread(void *arg)
{
        printf("This is worker_thread()\n");
        pthread_exit(NULL);
}

int main()
{
        pthread_t my_thread;
        int ret;

        printf("In main: creating a thread\n");
        ret =  pthread_create(&my_thread, NULL, &worker_thread, NULL);
        if(ret != 0) {
                printf("Error: pthread_create() failed\n");
                exit(EXIT_FAILURE);
        }

        pthread_exit(NULL);
}
```

Stopping the Threads : pthread_join



```
/* For each row of A */
for (i = 0; i < m; i++) {
    y[i] = 0.0;
    /* For each element of the row and each element of x */
    for (j = 0; j < n; j++)
        y[i] += A[i][j]* x[j];
}
```

$$y_i = \sum_{j=0}^{n-1} a_{ij} x_j$$

```
/* For each row of A */
for (i = 0; i < m; i++) {
    y[i] = 0.0;
    /* For each element of the row and each element of x */
    for (j = 0; j < n; j++)
        y[i] += A[i][j]* x[j];
}
```

$$y_i = \sum_{j=0}^{n-1} a_{ij} x_j$$

Using 3 Pthreads

Thread	Components of y
0	y[0], y[1]
1	y[2], y[3]
2	y[4], y[5]

thread 0

```
y[0] = 0.0;
for (j = 0; j < n; j++)
   y[0] += A[0][j]* x[j];
```

general case

```
y[i] = 0.0;
for (j = 0; j < n; j++)
   y[i] += A[i][j]*x[j];
```

```
void *Pth_mat_vect(void* rank) {
   long my_rank = (long) rank;
   int i, j;
   int local_m = m/thread_count;
   int my_first_row = my_rank*local_m;
   int my_last_row = (my_rank+1)*local_m - 1;

   for (i = my_first_row; i <= my_last_row; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
         y[i] += A[i][j]*x[j];
   }

   return NULL;
} /* Pth_mat_vect */
```

Ex:02: Thread with shared data

```
#include <stdio.h>
#include <assert.h>
#include <pthread.h>

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: end\n");
    return 0;
}
```

Thread with shared data


```
static volatile int counter = 0;

//
// mythread()
//
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
void *
mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}
```

```
//
// main()
//
// Just launches two threads (pthread_create)
// and then waits for them (pthread_join)
//
int
main(int argc, char *argv[])
{
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

Thread with shared data


```
prompt> gcc -o main main.c -Wall -pthread
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 20000000)
```

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

CRITICAL SECTIONS

- Matrix-vector multiplication:
  - Shared-memory locations were accessed in a simple way.
  - Threads make changes to y: but elements are owned by a thread
  - Does Multiple threads to modify the same element ?

- **Thread with shared data**
  - Multiple threads to modify the same.
  - Incorrect/different results if there is no synchronization

What happens when multiple threads update a single memory location

Race Condition

A race condition or data race occurs when

two processors (or two threads) access the same variable, and at least one does a write

The accesses are concurrent (not synchronized) so they could happen simultaneously

Race Condition and Synchronization

- **Critical Section**
  - **Portion of code that lead to race condition.**
  - **How to avoid the race condition ?**
    - **Mutual exclusion**
      - **Only on thread should execute critical section (Atomicity)**
    - **How to Mutual exclusion achieved?**
      - **Use of Locks**

Synchronization ?

- **Busy-Waiting**
  - A thread repeatedly tests a condition, but, effectively, does no useful work until the condition has the appropriate value.
  - Strict Alteration :
    - Use of int
    - [Thread 1 Continuou

Process 0

```
While (TRUE)
{
        while (turn != 0);
        critical_region();
        turn = 1;
        noncritical_region();
}
```

Process 1

```
While (TRUE)
{
        while (turn != 1);
        critical_region();
        turn = =;
        noncritical_region();
}
```

Synchronization Primitives in Pthreads,

- Controlling Thread and Synchronization Attributes,
- Thread Cancellation,
- Composite Synchronization Cons

Synchronization Primitives in Pthreads,

- Critical segments in Threads are implemented using mutex lock
- Mutex-locks have two states: locked and unlocked. At any point of time, only one thread can lock a mutex lock. A lock is an atomic operation
- A thread entering a critical segment first tries to get a lock. It goes ahead when the lock is granted.

- Mutexes (locks)
  - Mutex (mutual exclusion) is a special type of variable that can be used to restrict access to a critical section to a single thread at a time.
  - A thread that is busy-waiting may continually use the CPU accomplishing nothing.
  - When the lock is set, no other thread can access the locked region of code. This Ensures synchronized access of shared resources in the code.
  - Mutex lock will only be released by the thread who locked it

Mutexes (locks)

The Pthreads API provides the following functions for handling mutex-locks:

```
int pthread_mutex_lock (
    pthread_mutex_t    *mutex_lock);
```

Locks a mutex object

thread waits for the mutex to become available

If successful, pthread_mutex_lock() returns 0.

If unsuccessful, pthread_mutex_lock() returns -

```
int pthread_mutex_init (
    pthread_mutex_t    *mutex_lock,
    const pthread_mutexattr_t    *lock_attr);
```

Creates a mutex, referenced by mutex, with attributes specified by attr.

pthread_mutex_init() returns 0

```
int pthread_mutex_unlock (
    pthread_mutex_t *mutex_lock);
```

Releases a mutex object.

If successful, pthread_mutex_unlock() returns 0.

If unsuccessful, pthread_mutex_unlock() returns -1

Ex:02: Thread with shared data

Consider:

```
/* each thread to update shared variable
   best_cost */

if (my_cost < best_cost)
   best_cost = my_cost;
```

- two threads,
- the initial value of `best_cost` is 100,
- the values of `my_cost` are 50 and 75 for threads t1 and t2

The value of best_cost could be 50 or 75!

The value 75 does not correspond to any serialization of the two threads

Ex:02: Thread with shared data

Use of mutex :

```
pthread_mutex_t cost_lock;
int main() {
        ...
   pthread_mutex_init(&cost_lock, NULL);
   pthread_create(&thhandle, NULL, find_best, ...)
        ...
}
void *find_best(void *list_ptr) {
   ...
   pthread_mutex_lock(&cost_lock); // enter CS
   if (my_cost < best_cost)
      best_cost = my_cost;          Critical Section
   pthread_mutex_unlock(&cost_lock); /
}
```

Producer-Consumer Using Locks

- The producer threads
  - must not overwrite the shared buffer when the previous task has not been picked up by a consumer thread.
- The consumer threads
  - must not pick up tasks until there is something present in the shared data structure.
  - Individual consumer thread should pick up tasks one at a Time

Contentions

–Between producers

– Between consumers

Between producers and consumers

Producer-Consumer Using Locks


```
pthread_mutex_t task_queue_lock;
int task_available;
main() {
    ....
    task_available = 0;
    pthread_mutex_init(&task_queue_lock, NULL);
    ....
}
```

```
void *producer(void *producer_thread_data) {
    ....
    while (!done()) {
        inserted = 0;
        create_task(&my_task);
        while (inserted == 0) {
            pthread_mutex_lock(&task_queue_lock);
            if (task_available == 0) {
                insert_into_queue(my_task);
                task_available = 1; inserted = 1;
            }
            pthread_mutex_unlock(&task_queue_lock)
        }
    }
}
```

Note the purpose of inserted and extraced variables

```
void *consumer(void *consumer_thread_data) {
    int extracted;
    struct task my_task;
    while (!done()) {
        extracted = 0;
        while (extracted == 0) {
            pthread_mutex_lock(&task_queue_lock);
            if (task_available == 1) {
                extract_from_queue(&my_task);
                task_available = 0; extracted = 1;
            }
            pthread_mutex_unlock(&task_queue_lock);
        }
        process_task(my_task);
    }
}
```

- Types of Mutexes
  - e type of the mutex can be set in the attributes object during initialization
  - Overheads of Locking Normal:
  - Recursive: recursive mutex allows a single thread to lock a mutex as many times as it want
  - error-check : eports an error when a thread with a lock tries to lock it again
  - – pthread_mutex_attr_init

Overheads of Locking :

- Encapsulating large segments leads to significant performance degradation.
- Reduce the blocking overhead associated with locks using: Mutex_trylock

```
int pthread_mutex_trylock (
      pthread_mutex_t *mutex_lock);
```

- acquire lock if available
- return EBUSY if not available – enables a thread to do something else if lock unavailable
- pthread trylock typically much faster than lock on certain systems

Condition Variables for Synchronization

- block a thread within a critical section until some condition is satisfied.
- A condition variable allows a thread to block itself until specified data reaches a predefined state.
- A condition variable is associated with this predicate. When the predicate becomes true, the condition variable is used to signal one or more threads waiting on the condition.
- A single condition variable may be associated with more than one predicate.
- A condition variable always has a mutex associated with it. A thread locks this mutex and tests the predicate defined on the shared variable.
- If the **predicate is not true**, the thread waits on the condition variable associated with the predicate using the **function pthread cond wait.**

Condition Variables for Synchronization :

- wait (*pthread_cond_wait()*)
- signal (*pthread_cond_signal()*)
- broadcast (*pthread_cond_broadcast()*)

```
int pthread_cond_wait(pthread_cond_t *cond,
    pthread_mutex_t *mutex);

int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_broadcast(pthread_cond_t *cond);

int pthread_cond_init(pthread_cond_t *cond,
    const pthread_condattr_t *attr);

int pthread_cond_destroy(pthread_cond_t *cond);
```

Example:

```
pthread_mutex_lock( &m );
. . .
while (!arbitrary_condition) {
    pthread_cond_wait( &cv, &m );
    }
. . .
pthread_mutex_unlock( &m );
```

- the mutex is acquired before the condition is tested.
- This ensures that only this thread has access to the arbitrary condition being examined.
- While the condition is true, the code sample will block on the wait call until some **other** thread performs a signal or broadcast on the condvar.

Producer-Consumer Using Condition Variables


```
pthread_cond_t cond_queue_empty, cond_queue_full;
pthread_mutex_t task_queue_cond_lock;
int task_available;

/* other data structures here */

main() {
    /* declarations and initializations */
    task_available = 0;
    pthread_init();
    pthread_cond_init(&cond_queue_empty, NULL);
    pthread_cond_init(&cond_queue_full, NULL);
    pthread_mutex_init(&task_queue_cond_lock, NULL);
    /* create and join producer and consumer threads */
}
```

Producer-Consumer Using Condition Variables


```
void *producer(void *producer_thread_data) {
    int inserted;
    while (!done()) {
        create_task();
        pthread_mutex_lock(&task_queue_cond_lock);
        while (task_available == 1)
            pthread_cond_wait(&cond_queue_empty,
                &task_queue_cond_lock);
        insert_into_queue();
        task_available = 1;
        pthread_cond_signal(&cond_queue_full);
        pthread_mutex_unlock(&task_queue_cond_lock);
    }
}
```

Producer-Consumer Using Condition Variables


```
void *consumer(void *consumer_thread_data) {
    while (!done()) {
        pthread_mutex_lock(&task_queue_cond_lock);
        while (task_available == 0)
            pthread_cond_wait(&cond_queue_full,
                &task_queue_cond_lock);
        my_task = extract_from_queue();
        task_available = 0;
        pthread_cond_signal(&cond_queue_empty);
        pthread_mutex_unlock(&task_queue_cond_lock);
        process_task(my_task);
    }
}
```

Controlling Thread and Synchronization Attributes

- The Pthreads API allows a programmer to change the default attributes of entities using attributes objects.
- An attributes object is a data-structure that describes entity (thread, mutex, condition variable) properties.
- Once these properties are set, the attributes object can be passed to the method initializing the entity.
- Enhances modularity, readability, and ease of modification

Attributes Objects for Threads

- Use `pthread_attr_init` to create an attributes object.
- Individual properties associated with the attributes object can be changed using the following functions: `pthread_attr_setdetachstate`, `pthread_attr_setguardsize_np`, `pthread_attr_setstacksize`, `pthread_attr_setinheritsched`, `pthread_attr_setschedpolicy`, and `pthread_attr_setschedparam`.

Attributes Objects for Mutexes

```
pthread_mutexattr_settype_np (
    pthread_mutexattr_t   *attr,
    int   type);
```

Here, `type` specifies the type of the mutex and can take one of:

- `PTHREAD_MUTEX_NORMAL_NP`
- `PTHREAD_MUTEX_RECURSIVE_NP`
- `PTHREAD_MUTEX_ERRORCHECK_NP`

Composite Synchronization Constructs

- Higher level constructs can be built using basic synchronization constructs.
  - Read-write locks
  - Barriers.

Composite Synchronization Constructs

- **Read-Write Locks**
  - **Multiple read allowed but no multiple write.**
  - A read lock is granted when there are other threads that may already have read locks. (multiple read lock)
  - If there is a write lock on the data (or if there are queued write locks), the thread performs a condition wait.
  - If there are multiple threads requesting a write lock, they must perform a condition wait.(Only one write at time)

Let us Design functions for read locks mylib_rwlock_rlock, write locks mylib_rwlock_wlock, and unlocking mylib_rwlock_unlock

Read-Write Locks

The lock data type mylib rwlock t holds the following: –

- a count of the number of readers,
- the writer (a 0/1 integer specifying whether a writer is present),
- a **condition variable** readers proceed that is signaled when readers can proceed
- a condition variable writer proceed that is signaled when one of the writers can proceed,
- a count pending writers of pending writers, and – a mutex read write lock associated with the shared data structure.

```
typedef struct {
    int readers;
    int writer;
    pthread_cond_t readers_proceed;
    pthread_cond_t writer_proceed;
    int pending_writers;
    pthread_mutex_t read_write_lock;
} mylib_rwlock_t;


void mylib_rwlock_init (mylib_rwlock_t *l) {
   l -> readers = l -> writer = l -> pending_writers = 0;
   pthread_mutex_init(&(l -> read_write_lock), NULL);
   pthread_cond_init(&(l -> readers_proceed), NULL);
   pthread_cond_init(&(l -> writer_proceed), NULL);
}
```

```
void mylib_rwlock_rlock(mylib_rwlock_t *l) {
   /* if there is a write lock or pending writers, perform condition
   wait.. else increment count of readers and grant read lock */

   pthread_mutex_lock(&(l -> read_write_lock));
   while ((l -> pending_writers > 0) || (l -> writer > 0))
       pthread_cond_wait(&(l -> readers_proceed),
          &(l -> read_write_lock));
   l -> readers ++;
   pthread_mutex_unlock(&(l -> read_write_lock));
}
```

```
void mylib_rwlock_wlock(mylib_rwlock_t *l) {
   /* if there are readers or writers, increment pending writers
   count and wait. On being woken, decrement pending writers
   count and increment writer count */

   pthread_mutex_lock(&(l -> read_write_lock));
   while ((l -> writer > 0) || (l -> readers > 0))  {
       l -> pending_writers ++;
       pthread_cond_wait(&(l -> writer_proceed),
          &(l -> read_write_lock));
   }
   l -> pending_writers --;
   l -> writer ++
   pthread_mutex_unlock(&(l -> read_write_lock));
}
```

```
void mylib_rwlock_unlock(mylib_rwlock_t *l) {
   /* if there is a write lock then unlock, else if there are
   read locks, decrement count of read locks. If the count
   is 0 and there is a pending writer, let it through, else
   if there are pending readers, let them all go through */

   pthread_mutex_lock(&(l -> read_write_lock));
   if (l -> writer > 0)
      l -> writer = 0;
   else if (l -> readers > 0)
      l -> readers --;
   pthread_mutex_unlock(&(l -> read_write_lock));
   if ((l -> readers == 0) && (l -> pending_writers > 0))
      pthread_cond_signal(&(l -> writer_proceed));
   else if (l -> readers > 0)
      pthread_cond_broadcast(&(l -> readers_proceed));
}
```

Barriers

- As in MPI, a barrier holds a thread until all threads participating in the barrier have reached it.
- Barriers can be implemented using a counter, a mutex and a condition variable.
- A single integer is used to keep track of the number of threads that have reached the barrier.
- If the count is less than the total number of threads, the threads execute a condition wait.
- The last thread entering (and setting the count to the number of threads) wakes up all the threads using a condition broadcast.

```
typedef struct {
    pthread_mutex_t count_lock;
    pthread_cond_t ok_to_proceed;
    int count;
} mylib_barrier_t;

void mylib_init_barrier(mylib_barrier_t *b) {
    b -> count = 0;
    pthread_mutex_init(&(b -> count_lock), NULL);
    pthread_cond_init(&(b -> ok_to_proceed), NULL);
}

void mylib_barrier (mylib_barrier_t *b, int num_threads) {
    pthread_mutex_lock(&(b -> count_lock));
    b -> count ++;
    if (b -> count == num_threads) {
        b -> count = 0;
        pthread_cond_broadcast(&(b -> ok_to_proceed));
    }
    else
        while (pthread_cond_wait(&(b -> ok_to_proceed),
            &(b -> count_lock)) != 0);
    pthread_mutex_unlock(&(b -> count_lock));
}
```

- Barrier, threads enter the barrier and stay until the broadcast signal releases them.
- The threads are released one by one since the mutex *count_lock* is passed among them one after the other.