Concurrent Systems

3D4 ⟷ CS2016

Operating Systems

Andrew Butterfield
ORI.G39, Andrew.Butterfield@scss.tcd.ie

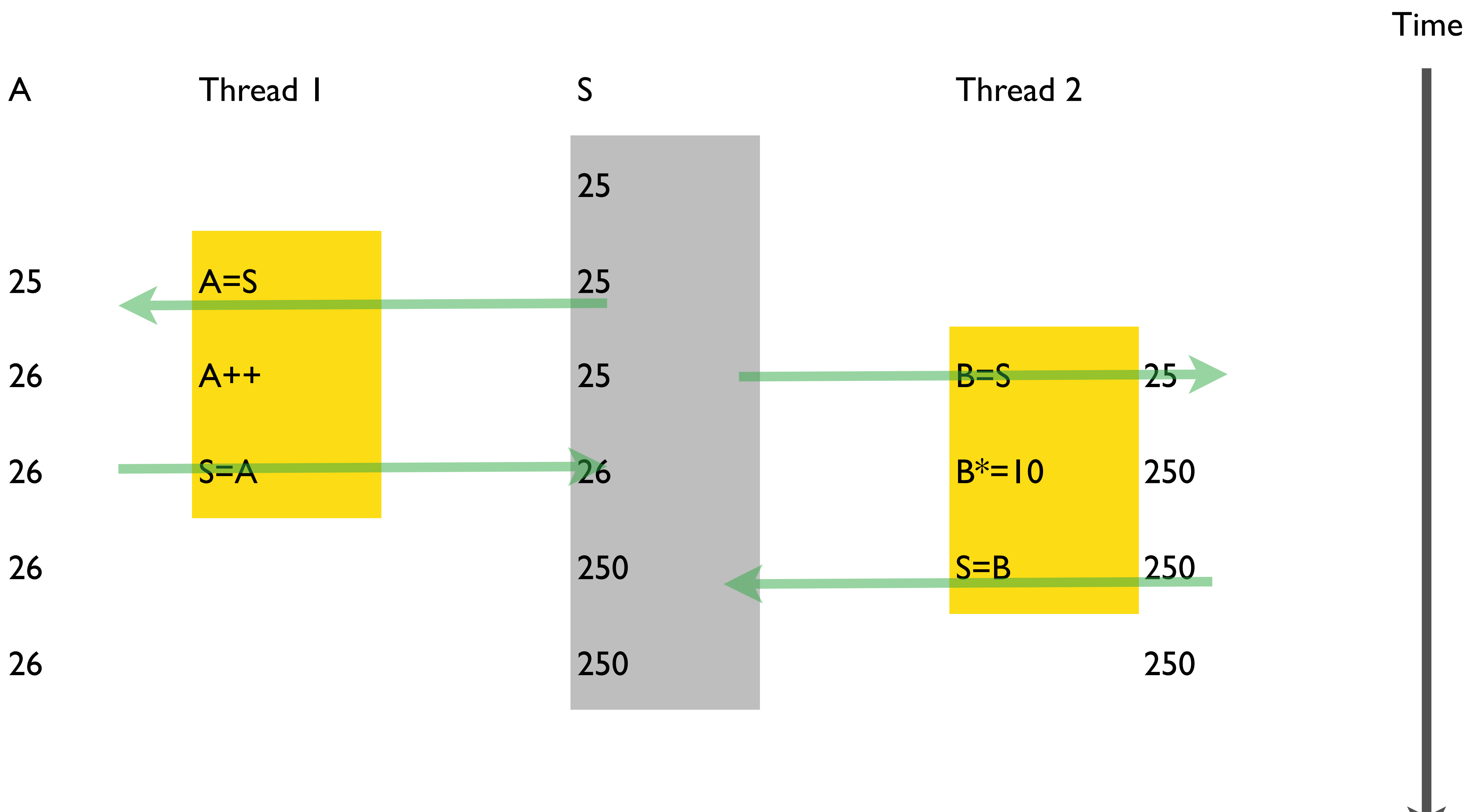
Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

with thanks to Mike Brady

Synchronisation

- We’ve looked at situations where the threads can operate independently -- the ‘write sets’ of the threads don’t intersect.
- Where the write sets intersect, we must ensure that independent thread writes do not damage the data.

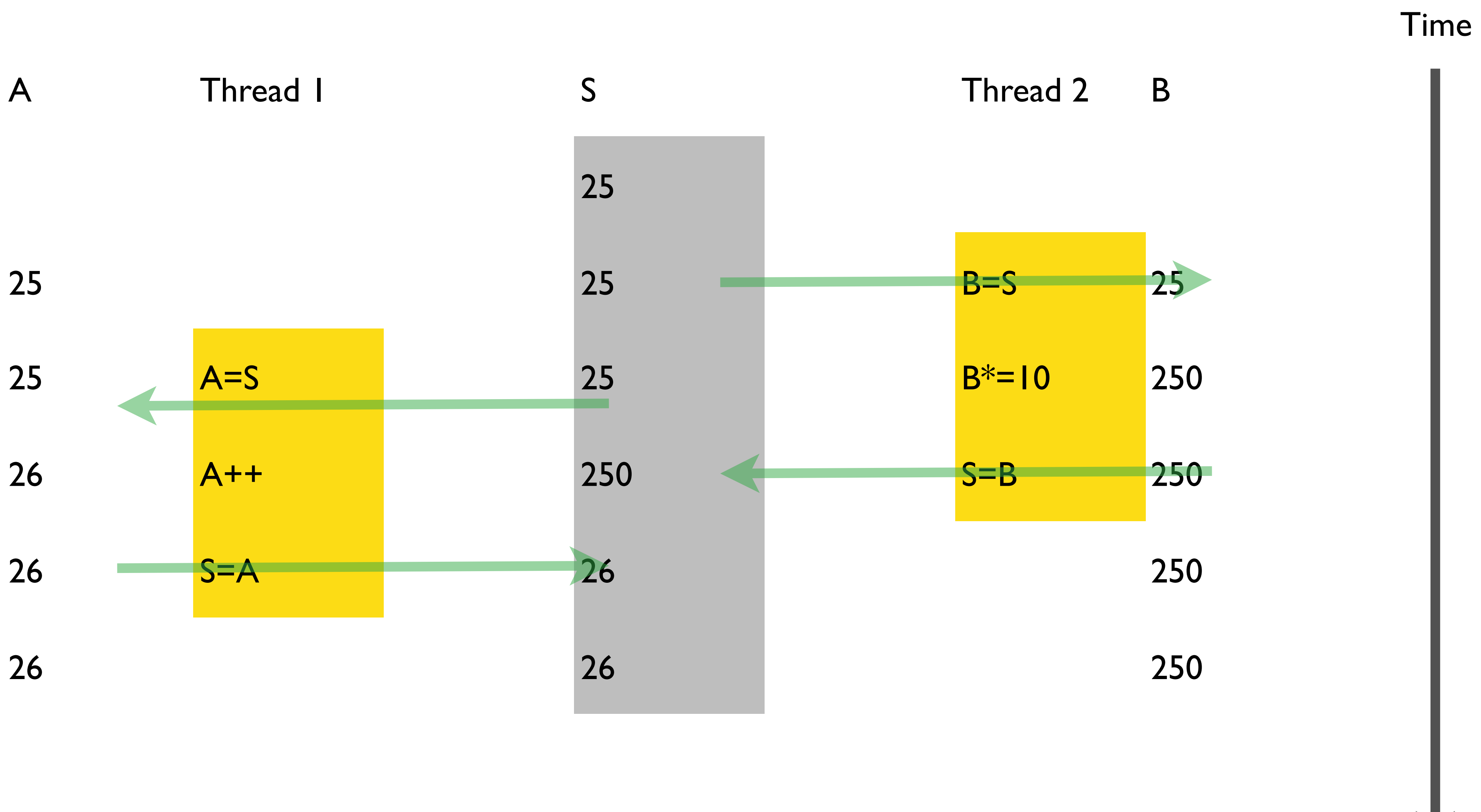
Shared Variable S: Thread 1 *before* Thread 2



A	Thread 1	S	Thread 2	
		25		
25	A=S	25		
26	A++	25	B=S	25
26	S=A	26	B*=10	250
26		250	S=B	250
26		250		250

Thread 2 accesses S after Thread 1 but before Thread 1 has finished updating S

Shared Variable S: Thread 1 *after* Thread 2



A	Thread 1	S	Thread 2	B
		25		
25		25	B=S	25
25	A=S	25	B*=10	250
26	A++	250	S=B	250
26	S=A	26		250
26		26		250

Thread 1 accesses S after Thread 2 but before Thread 2 has finished modifying S

Synchronisation

- Problem: Access to shared resources can be dangerous.
  - These are so-called ‘critical’ accesses.
- Solution. Critical accesses should be made exclusive. Thus, all critical accesses to a resource are *mutually exclusive*.
- In the example, both threads should have asked for exclusive access before making their updates.
  - Depending on timing, one or the other would get exclusive access first. The other would have to wait to get any kind of access.

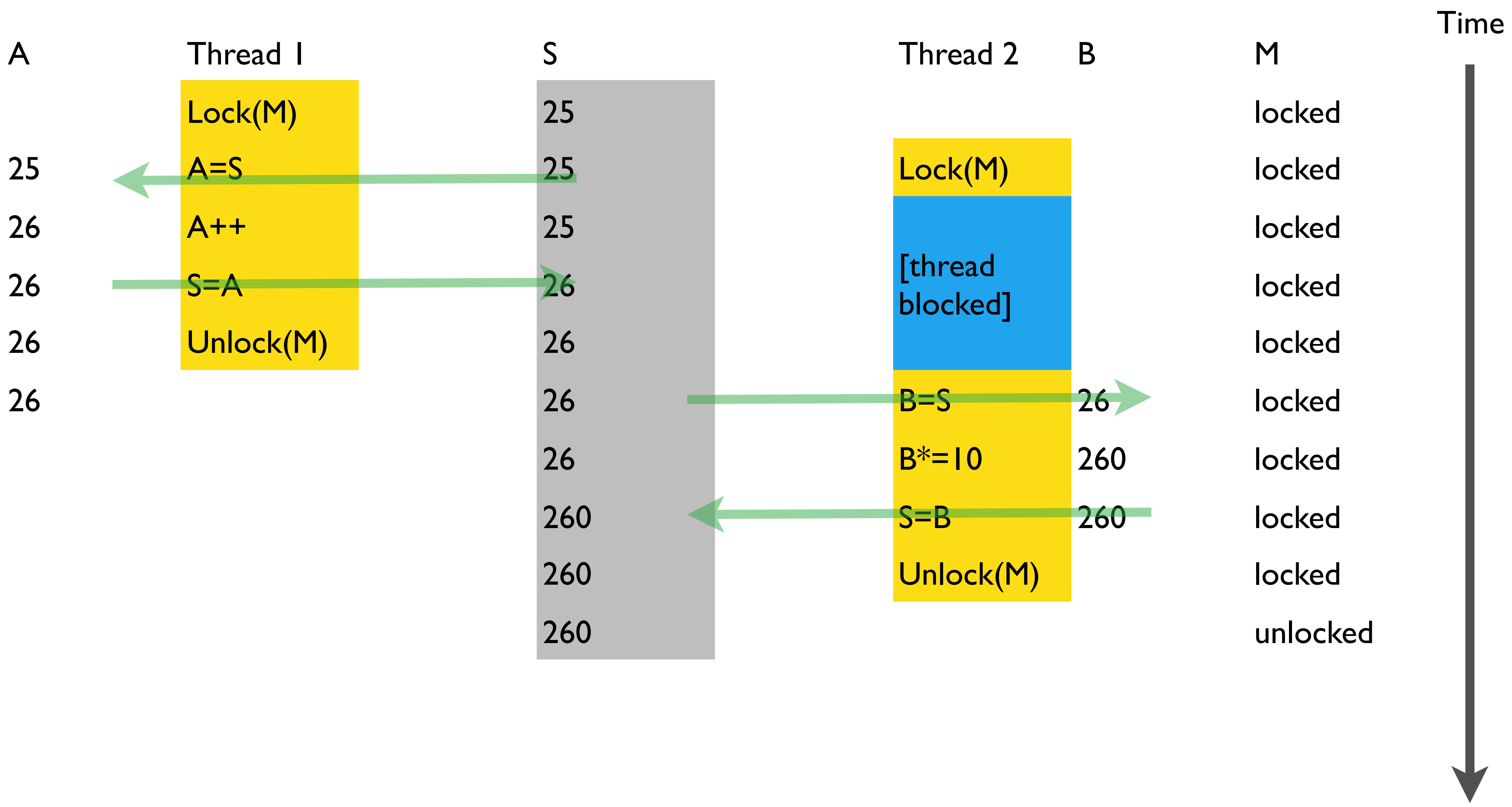
Mutual Exclusion in pthreads.

- Mutual Exclusion is accomplished using ‘*mutex*’ variables.
- Mutex variables are used to protect access to a resource.

Accessing a protected resource

- To access a mutex-protected resource, an agent must acquire a lock on the mutex variable.
  - If the mutex variable is unlocked, it is immediately locked and the agent has acquired it. When finished, the agent must unlock it.
  - If the mutex variable is already locked, the agent has failed to acquire the lock -- the protected resource is in exclusive use by someone else.
    - The agent is usually blocked until lock is acquired.
    - A non-blocking version of lock acquisition is available.

Shared Variable S Protected by Mutex M



A	Thread 1	S	Thread 2	B	M
	Lock(M)	25			locked
25	A=S	25	Lock(M)		locked
26	A++	25	[thread blocked]		locked
26	S=A	26			locked
26	Unlock(M)	26			locked
26		26	B=S	26	locked
		26	B*=10	260	locked
		260	S=B	260	locked
		260	Unlock(M)		locked
		260			unlocked

Create pthread mutex variable

- Static:
  - pthread_mutex_t m = PTHREAD_MUTEX_INITIALISER;
    - Initially unlocked
- Dynamic
  - pthread_mutex_init(<ref to mutex variable>,attributes)

Lock and Unlock Mutex

- pthread_mutex_lock(<mutex variable reference>);
  - acquire lock or block while waiting
- pthread_mutex_trylock(<mutex variable reference>);
  - non-blocking; check returned code
- pthread_mutex_unlock(<mutex variable reference>);

Example (1 of 2) – Thread

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

#define checkResults(string, val) {             \
if (val) {                                     \
printf("Failed with %d at %s", val, string); \
exit(1);                                     \
}                                              \
}

// From http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp?topic=/rzahw/rzahwe18rx.htm
// Fixed to avoid calls to non-standard pthread_getthreadid_np()

#define                 NUMTHREADS   3
pthread_mutex_t         mutex = PTHREAD_MUTEX_INITIALIZER;
int                     sharedData=0;
int                     sharedData2=0;

void *theThread(void *threadid)
{
    int   rc;
    printf("Thread %.8x: Entered\n", (int)threadid);
    rc = pthread_mutex_lock(&mutex);
    checkResults("pthread_mutex_lock()\n", rc);

    /********** Critical Section *******************/
    printf("Thread %.8x: Start critical section, holding lock\n",(int)threadid);
    /* Access to shared data goes here */
    ++sharedData; --sharedData2;
    printf("Thread %.8x: End critical section, release lock\n",(int)threadid);
    /********** Critical Section *******************/

    rc = pthread_mutex_unlock(&mutex);
    checkResults("pthread_mutex_unlock()\n", rc);
    return NULL;
}
```

Example (2 of 2) – Main

```
int main(int argc, char **argv)
{
    pthread_t             thread[NUMTHREADS];
    int                   rc=0;
    int                   i;

    printf("Enter Testcase - %s\n", argv[0]);

    printf("Hold Mutex to prevent access to shared data\n");
    rc = pthread_mutex_lock(&mutex);
    checkResults("pthread_mutex_lock()\n", rc);

    printf("Create/start threads\n");
    for (i=0; i<NUMTHREADS; ++i) {
        rc = pthread_create(&thread[i], NULL, theThread, (void *)i);
        checkResults("pthread_create()\n", rc);
    }

    printf("Wait a bit until we are 'done' with the shared data\n");
    sleep(3);
    printf("Unlock shared data\n");
    rc = pthread_mutex_unlock(&mutex);
    checkResults("pthread_mutex_lock()\n",rc);

    printf("Wait for the threads to complete, and release their resources\n");
    for (i=0; i <NUMTHREADS; ++i) {
        rc = pthread_join(thread[i], NULL);
        checkResults("pthread_join()\n", rc);
    }

    printf("Results: sharedData: %d, sharedData2: %d\n",sharedData,sharedData2);

    printf("Clean up\n");
    rc = pthread_mutex_destroy(&mutex);
    printf("Main completed\n");
    return 0;
}
```

Problems

- Voluntary
  - Mutexes ‘protect’ *code*.
  - Other programmers don’t have to use them to get access to the variables the code accesses.
    - This is part of the tradeoff. Use processes rather than threads if you want better protection.
- Unfair
  - If multiple threads are blocked on a mutex, the order in which they waken up is not guaranteed to be any particular order.