OSD 19CS2106S
Session – 36
Mutex and Concurrent Linked Lists
Lecture Notes

```
/*   Concurrent Linked Lists  */
#include <stdio.h>
#include <stdlib.h>
typedef struct __node_t {
    int             key;
    struct __node_t        *next;
} node_t;
// basic list structure (one used per list)
typedef struct __list_t {
node_t *head;
pthread_mutex_t lock;
} list_t;
void List_Init(list_t *L) {
    L->head = NULL;
    pthread_mutex_init(&L->lock, NULL);
}
int List_Insert(list_t *L, int key) {
    pthread_mutex_lock(&L->lock);
       node_t *new = malloc(sizeof(node_t));
       if (new == NULL) {
              perror("malloc");
              pthread_mutex_unlock(&L->lock);
              return -1; // fail
       }
       new->key = key;
       new->next = L->head;
       L->head = new;
       pthread_mutex_unlock(&L->lock);
       return 0; // success
}
int List_Lookup(list_t *L, int key) {
       pthread_mutex_lock(&L->lock);
       node_t *curr = L->head;
       while (curr) {
              if (curr->key == key) {
              pthread_mutex_unlock(&L->lock);
              return 0; // success
              }
       curr = curr->next;
       }
       pthread_mutex_unlock(&L->lock);
       return -1; // failure
}
void List_Print(list_t *L) {
    node_t *tmp = L->head;
    while (tmp) {
       printf("%d ", tmp->key);
       tmp = tmp->next;
    }
    printf("\n");
}
int main(int argc, char *argv[])
{
    list_t mylist;
    List_Init(&mylist);
    List_Insert(&mylist, 10);
    List_Insert(&mylist, 30);
    List_Insert(&mylist, 5);
    List_Print(&mylist);
    printf("In List: 10? %d 20? %d\n",
         List_Lookup(&mylist, 10), List_Lookup(&mylist, 20));
    return 0;
}
/*
vishnu@mannava:~/threads$ ./a.out;./a.out
5 30 10
In List: 10? 0 20? -1
5 30 10
In List: 10? 0 20? -1
*/
```

- The code **acquires** a lock in the insert routine upon entry.
- The code **releases** the lock upon exit.

- If malloc() happens to *fail*, the code must also release the lock before failing the insert.
- This kind of exceptional control flow has been shown to be quite error prone.
- **Solution:** The lock and release *only surround* the actual critical section in the insert code

Concurrent Linked List: Rewritten

```
1     void List_Init(list_t *L) {
2           L->head = NULL;
3           pthread_mutex_init(&L->lock, NULL);
4     }
5
```

```
6      void List_Insert(list_t *L, int key) {
7            // synchronization not needed
8            node_t *new = malloc(sizeof(node_t));
9            if (new == NULL) {
10                 perror("malloc");
11                 return;
12           }
13           new->key = key;
14
15           // just lock critical section
16           pthread_mutex_lock(&L->lock);
17           new->next = L->head;
18           L->head = new;
19           pthread_mutex_unlock(&L->lock);
20     }
```

```
21
22     int List_Lookup(list_t *L, int key) {
23           int rv = -1;
24           pthread_mutex_lock(&L->lock);
25           node_t *curr = L->head;
26           while (curr) {
27                 if (curr->key == key) {
28                       rv = 0;
29                       break;
30                 }
31                 curr = curr->next;
32           }
33           pthread_mutex_unlock(&L->lock);
34           return rv; // now both success and failure
35     }
```

Scaling Linked List:

- Hand-over-hand locking (lock coupling)
  - Add **a lock per node** of the list instead of having a single lock for the entire list.
  - When traversing the list,
    - First grabs the next node's lock.
    - And then releases the current node's lock.
  - Enable a high degree of concurrency in list operations.

However, in practice, the overheads of acquiring and releasing locks for each node of a list traversal is *prohibitive*.

Pthreads Read-Write Locks

Neither of our multi-threaded linked lists exploits the potential for simultaneous access to any node by threads that are executing Member.
The **first solution** only allows one thread to access the entire list at any instant
The **second** only allows one thread to access any given node at any instant.
A read-write lock is somewhat like a mutex except that it provides two lock functions.
The first lock function locks the read-write lock for reading, while the second locks it for writing.
So multiple threads can simultaneously obtain the lock by calling the read-lock function, while only one thread can obtain the lock by calling the write-lock function.
Thus, if any threads own the lock for reading, any threads that want to obtain the lock for writing will block in the call to the write-lock function.
If any thread owns the lock for writing, any threads that want to obtain the lock for reading or writing will block in their respective locking functions.
Readerwriter locks are similar to mutexes, except that they allow for higher degrees of parallelism. With a mutex, the state is either locked or unlocked, and only one thread can lock it at a time. Three states are possible with a readerwriter lock: locked in read mode, locked in write mode, and unlocked. Only one thread at a time can hold a readerwriter lock in write mode, but multiple threads can hold a readerwriter lock in read mode at the same time.
When a readerwriter lock is write-locked, all threads attempting to lock it block until it is unlocked. When a readerwriter lock is read-locked, all threads attempting to lock it in read mode are given access, but any threads attempting to lock it in write mode block until all the threads have relinquished their read locks. Although implementations vary, readerwriter locks usually block additional readers if a lock is already held in read mode and a thread is blocked trying to acquire the lock in write mode. This prevents a constant stream of readers from starving waiting writers.
Readerwriter locks are well suited for situations in which data structures are read more often than they are modified. When a readerwriter lock is held in write mode, the data structure it protects can be modified safely, since only one thread at a time can hold the lock in write mode. When the readerwriter lock is held in read mode, the data structure it protects can be read by multiple threads, as long as the threads first acquire the lock in read mode.
Readerwriter locks are also called sharedexclusive locks. When a readerwriter lock is read-locked, it is said to be locked in shared mode. When it is write-locked, it is said to be locked in exclusive mode.
As with mutexes, readerwriter locks must be initialized before use and destroyed before freeing their underlying memory.

```
#include <pthread.h>
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock, const pthread_rwlockattr_t *restrict attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
Both return: 0 if OK, error number on failure
#include <pthread.h>
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);          All return: 0 if OK, error number on failure
```

```
/* Pthreads - Increment Problem without
Synchronization*/
#include <pthread.h>
#include<stdio.h>
static int glob = 0;
/* Loop 'arg' times incrementing 'glob' */
static void *  threadFunc(void *arg)
{
   int loops = *((int *) arg);
   int loc, j;
   for (j = 0; j < loops; j++) {
      loc = glob;
      loc++;
      glob = loc;
   }
   return NULL;
}
Int main(int argc, char *argv[])
{
   pthread_t t1, t2;
   long loops, s;
   loops = strtol(argv[1], NULL, 10);
   s = pthread_create(&t1, NULL, threadFunc,
&loops);
   if (s != 0)
      perror("pthread_create");
   s = pthread_create(&t2, NULL, threadFunc,
&loops);
   if (s != 0)
      perror("pthread_create");
   s = pthread_join(t1, NULL);
   if (s != 0)
      perror("pthread_join");
   s = pthread_join(t2, NULL);
   if (s != 0)
      perror("pthread_join");
   printf("glob = %d\n", glob);
   exit(0);
}
/*
 [vishnu@mannava PP]$ ./a.out 500
glob = 1000
[vishnu@mannava PP]$ ./a.out 50000
glob = 51227
[vishnu@mannava PP]$ ./a.out 5000000
glob = 6268601
[vishnu@mannava PP]$ ./a.out 50000000
glob = 59328262
*/
```

```
/* Pthreads - Increment Problem with Mutex -
Synchronization */
#include <pthread.h>
#include<stdio.h>
static int glob = 0;
static pthread_mutex_t mtx =
PTHREAD_MUTEX_INITIALIZER;
/* Loop 'arg' times incrementing 'glob' */
static void *  threadFunc(void *arg)
{
   int loops = *((int *) arg);
   int loc, j;
   for (j = 0; j < loops; j++) {
      pthread_mutex_lock(&mtx);
      loc = glob;
      loc++;
      glob = loc;
      pthread_mutex_unlock(&mtx);
   }
   return NULL;
}
int
main(int argc, char *argv[])
{
   pthread_t t1, t2;
   long loops, s;
   loops = strtol(argv[1], NULL, 10);
   s = pthread_create(&t1, NULL, threadFunc,
&loops);
   if (s != 0)
      perror("pthread_create");
   s = pthread_create(&t2, NULL, threadFunc,
&loops);
   if (s != 0)
      perror("pthread_create");
   s = pthread_join(t1, NULL);
   if (s != 0)
      perror("pthread_join");
   s = pthread_join(t2, NULL);
   if (s != 0)
      perror("pthread_join");
   printf("glob = %d\n", glob);
   exit(0);
}
/*
[vishnu@mannava PP]$ ./a.out 500
glob = 1000
[vishnu@mannava PP]$ ./a.out 50000
glob = 100000
[vishnu@mannava PP]$ ./a.out 5000000
glob = 10000000
[vishnu@mannava PP]$ ./a.out 50000000
glob = 100000000
*/
```

```
/* Pthreads - Parallel Global Sum with Synchronization- Final*/
#include<stdio.h>
#include<pthread.h>
#include<stdlib.h>
#define no_threads 10
#define n 20
static long sum = 0;
long a[n];
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static void *slave(void *st)
{
long rank=(long*)st;
long my_n=n/no_threads;
long my_first_i = my_n *rank;
long my_last_i = my_first_i + my_n;
long i;
for (i = my_first_i; i < my_last_i; i++)
  {
    pthread_mutex_lock(&mtx);
    printf(" thread rank = %d Global sum it is fetching is %d \n", rank, sum);
    sum += *(a + i);
    printf(" hello from thread %d and its partial sum = %d\n",rank,sum);
    pthread_mutex_unlock(&mtx);
  }
return NULL; }
main()
{
long i;
pthread_t thread[10];
for (i = 0; i < n; i++)
      a[i] = i+1;
for (i = 0; i < no_threads; i++)
{
if (pthread_create(&thread[i], NULL, slave,(void*) i) != 0)
perror("Pthread create fails");
}
for (i = 0; i < no_threads; i++) {
if (pthread_join(thread[i], NULL) != 0)
 perror("Pthread join fails");  }
printf("The sum of numbers from %d to %d is %d\n",a[0],a[n-1], sum);
exit(0); } /*
[vishnu@mannava PP]$ ./a.out
 thread rank = 0 Global sum it is fetching is 0
 hello from thread 0 and its partial sum = 1
 thread rank = 0 Global sum it is fetching is 1
 hello from thread 0 and its partial sum = 3
 thread rank = 8 Global sum it is fetching is 3
 hello from thread 8 and its partial sum = 20
 thread rank = 8 Global sum it is fetching is 20
 hello from thread 8 and its partial sum = 38
 thread rank = 3 Global sum it is fetching is 38
 hello from thread 3 and its partial sum = 45
 thread rank = 3 Global sum it is fetching is 45
 hello from thread 3 and its partial sum = 53
 thread rank = 5 Global sum it is fetching is 53
 hello from thread 5 and its partial sum = 64
 thread rank = 5 Global sum it is fetching is 64
 hello from thread 5 and its partial sum = 76
 thread rank = 1 Global sum it is fetching is 76
 hello from thread 1 and its partial sum = 79
 thread rank = 1 Global sum it is fetching is 79
 hello from thread 1 and its partial sum = 83
 thread rank = 9 Global sum it is fetching is 83
 hello from thread 9 and its partial sum = 102
 thread rank = 9 Global sum it is fetching is 102
 hello from thread 9 and its partial sum = 122
 thread rank = 4 Global sum it is fetching is 122
 hello from thread 4 and its partial sum = 131
 thread rank = 4 Global sum it is fetching is 131
 hello from thread 4 and its partial sum = 141
 thread rank = 7 Global sum it is fetching is 141
 hello from thread 7 and its partial sum = 156
 thread rank = 7 Global sum it is fetching is 156
 hello from thread 7 and its partial sum = 172
 thread rank = 6 Global sum it is fetching is 172
 hello from thread 6 and its partial sum = 185
 thread rank = 6 Global sum it is fetching is 185
 hello from thread 6 and its partial sum = 199
 thread rank = 2 Global sum it is fetching is 199
 hello from thread 2 and its partial sum = 204
 thread rank = 2 Global sum it is fetching is 204
 hello from thread 2 and its partial sum = 210
 The sum of numbers from 1 to 20 is 210 */
```