Advanced Topics on Shared Memory Programming with Pthreads

Pacheco. Chapter 4
T. Yang. UCSB CS140. Spring 2014
Outline

• More on thread synchronization.
  ▪ Read-write locks.
  ▪ Applications in a shared link list
• False sharing
• Deadlocks and thread safety.
READ-WRITE LOCKS
A data set is shared among a number of concurrent threads.
- Readers – only read the data set; they do not perform any updates
- Writers – can both read and write

Requirement:
- allow multiple readers to read at the same time.
- Only one writer can access the shared data at the same time.

Reader/Writer access permission table:

<table>
<thead>
<tr>
<th></th>
<th>Reader</th>
<th>Writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reader</td>
<td>OK</td>
<td>No</td>
</tr>
<tr>
<td>Writer</td>
<td>NO</td>
<td>No</td>
</tr>
</tbody>
</table>
Readers-Writers (First try with 1 mutex lock)

- **writer**
  ```
  do {
    mutex_lock(w);
    // writing is performed
    mutex_unlock(w);
  } while (TRUE);
  ```

- **Reader**
  ```
  do {
    mutex_lock(w);
    // reading is performed
    mutex_unlock(w);
  } while (TRUE);
  ```
Readers-Writers (First try with 1 mutex lock)

- **writer**
  ```c
  do {
    mutex_lock(w);
    // writing is performed
    mutex_unlock(w);
  } while (TRUE);
  ```

- **Reader**
  ```c
  do {
    mutex_lock(w);
    // reading is performed
    mutex_unlock(w);
  } while (TRUE);
  ```

---

<table>
<thead>
<tr>
<th></th>
<th>Reader</th>
<th>Writer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reader</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Writer</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
2\textsuperscript{nd} try using a lock + readcount

• **writer**

```c
    do {
        mutex_lock(w);// Use writer mutex lock
        //    writing is performed
        mutex_unlock(w);
    } while (TRUE);
```

• **Reader**

```c
    do {
        readcount++;// add a reader counter.
        if(readcount==1) mutex_lock(w);
        //    reading is performed
        readcount--;
        if(readcount==0) mutex_unlock(w);
    } while (TRUE);
```
Readers-Writers Problem with semaphore

• **Shared Data**
  - Data set
  - Lock `mutex` (to protect `readcount`)
  - Semaphore `wrt` initialized to 1 (to synchronize between readers/writers)
  - Integer `readcount` initialized to 0
Readers-Writers Problem

- A writer

  \[
  \text{do } \{
  \quad \text{sem\_wait}(\text{wrt}) \; ; \; //\text{semaphore wrt} \\
  \quad // \text{ writing is performed} \\
  \quad \text{sem\_post}(\text{wrt}) \; ; \; // \\
  \}
  \quad \text{while (TRUE)};
  \]
Readers-Writers Problem (Cont.)

- Reader
  
  do {
    mutex_lock(mutex);
    readcount ++ ;
    if (readcount == 1)
        sem_wait(wrt); //check if anybody is writing
    mutex_unlock(mutex)

    // reading is performed

    mutex_lock(mutex);
    readcount - - ;
    if (readcount == 0)
        sem_post(wrt) ; //writing is allowed now
    nlock(mutex) ;
  } while (TRUE);
Application case: Sharing a sorted linked list of integers

- Demonstrate controlling of access to a large, shared data structure
- Operations supported
  - Member, Insert, and Delete.

```c
struct list_node_s {
  int data;
  struct list_node_s* next;
}
```
Membership operation for a linked list

```c
int Member(int value, struct list_node_s* head_p) {
    struct list_node_s* curr_p = head_p;

    while (curr_p != NULL && curr_p->data < value)
        curr_p = curr_p->next;

    if (curr_p == NULL || curr_p->data > value) {
        return 0;
    } else {
        return 1;
    }
} /* Member */
```

```
head_p  →  2  →  5  →  8
```
Insert operation: Inserting a new node
Inserting a new node into a list

```c
int Insert(int value, struct list_node_s** head_pp) {
    struct list_node_s* curr_p = *head_pp;
    struct list_node_s* pred_p = NULL;
    struct list_node_s* temp_p;

    while (curr_p != NULL && curr_p->data < value) {
        pred_p = curr_p;
        curr_p = curr_p->next;
    }

    if (curr_p == NULL || curr_p->data > value) {
        temp_p = (struct list_node_s*) malloc(sizeof(struct list_node_s));
        temp_p->data = value;
        temp_p->next = curr_p;
        if (pred_p == NULL) /* New first node */
            *head_pp = temp_p;
        else
            pred_p->next = temp_p;
        return 1;
    } else { /* Value already in list */
        return 0;
    }
} /* Insert */
```
Delete operation: remove a node from a linked list
Deleting a node from a linked list

```c
int Delete(int value, struct list_node_s** head_pp) {
    struct list_node_s* curr_p = *head_pp;
    struct list_node_s* pred_p = NULL;

    while (curr_p != NULL && curr_p->data == value) {
        pred_p = curr_p;
        curr_p = curr_p->next;
    }

    if (curr_p != NULL && curr_p->data == value) {
        if (pred_p == NULL) { /* Deleting first node in list */
            *head_pp = curr_p->next;
            free(curr_p);
        } else {
            pred_p->next = curr_p->next;
            free(curr_p);
        }
        return 1;
    } else { /* Value isn’t in list */
        return 0;
    }
} /* Delete */
```

Find a node with the given value

Remove this node
A Multi-Threaded Linked List

- Allow a sorted linked list to be accessed by multiple threads
- In order to share access to the list, define head_p to be a global variable.
  - This will simplify the function headers for Member, Insert, and Delete,
  - since we won’t need to pass in either head_p or a pointer to head_p: we’ll only need to pass in the value of interest.
Simultaneous access by two threads

Thread 0: curr_p

Thread 1: pred_p
Thread 1: curr_p

head_p → 2 → 5 → 8
Solution #1

• An obvious solution is to simply lock the list any time that a thread attempts to access it.
• A call to each of the three functions can be protected by a mutex.

```c
Pthread_mutex_lock(&list_mutex);
Member(value);
Pthread_mutex_unlock(&list_mutex);
```

In place of calling Member(value).
Issues

- We’re serializing access to the list.
- If the vast majority of our operations are calls to Member, we’ll fail to exploit this opportunity for parallelism.
- On the other hand, if most of our operations are calls to Insert and Delete,
  - This may be the best solution—since serialization of infrequent operations has minimum performance impact.
  - Easy to implement.

<table>
<thead>
<tr>
<th>List-level</th>
<th>Member</th>
<th>Insert</th>
<th>Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Insert</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Delete</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Insert</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delete</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
Solution #2

- Instead of locking the entire list, lock individual nodes.
  - A “finer-grained” approach: One mutex lock per node

```c
struct list_node_s {
    int data;
    struct list_node_s* next;
    pthread_mutex_t mutex;
};
```

<table>
<thead>
<tr>
<th>Node-level</th>
<th>Member</th>
<th>Insert</th>
<th>Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Insert</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Delete</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
```c
int Member(int value) {
    struct list_node_s* temp_p;

    pthread_mutex_lock(&head_p_mutex);
    temp_p = head_p;
    while (temp_p != NULL && temp_p->data < value) {
        if (temp_p->next != NULL)
            pthread_mutex_lock(&temp_p->next->mutex);
        if (temp_p == head_p)
            pthread_mutex_unlock(&head_p_mutex);
        pthread_mutex_unlock(&temp_p->mutex);
        temp_p = temp_p->next;
    }
}
```
if (temp_p == NULL || temp_p->data > value) {
    if (temp_p == head_p)
        pthread_mutex_unlock(&head_p_mutex);
    if (temp_p != NULL)
        pthread_mutex_unlock(&(temp_p->mutex));
    return 0;
} else {
    if (temp_p == head_p)
        pthread_mutex_unlock(&head_p_mutex);
    pthread_mutex_unlock(&(temp_p->mutex));
    return 1;
}

/* Member */
Issues

• Much more complex than the original Member function.
• Much slower,
  ▪ each time a node is accessed, a mutex must be locked and unlocked.
• Significant space cost
  ▪ Adding a mutex field to each node
Motivation for using 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.
Pthreads Read-Write Locks

• 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.

• Example for a linked list

<table>
<thead>
<tr>
<th></th>
<th>Member</th>
<th>Insert</th>
<th>Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>Insert</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>Delete</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

```c
pthread_rwlock_rdlock(&rwlock);
Member(value);
```

```c
pthread_rwlock_unlock(&rwlock);
```

```c
...
```

```c
pthread_rwlock_wrlock(&rwlock);
Insert(value);
```

```c
pthread_rwlock_unlock(&rwlock);
```

```c
...
```

```c
pthread_rwlock_wrlock(&rwlock);
Delete(value);
```

```c
pthread_rwlock_unlock(&rwlock);
```
Pthreads Read-Write Locks

• 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.

• 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.

<table>
<thead>
<tr>
<th>List-level</th>
<th>Member</th>
<th>Insert</th>
<th>Delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Insert</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>Delete</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
A performance comparison of 3 implementations for a linked list

Total time in second for executing 100,000 operations.
99.9% Member
0.05% Insert
0.05% Delete

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Number of Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Read-Write Locks</td>
<td>0.213</td>
</tr>
<tr>
<td>One Mutex for Entire List</td>
<td>0.211</td>
</tr>
<tr>
<td>One Mutex per Node</td>
<td>1.680</td>
</tr>
</tbody>
</table>
Linked List Performance: Comparison

Total time in seconds for executing 100,000 operations
80% Member
10% Insert
10% Delete

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Number of Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Read-Write Locks</td>
<td>2.48</td>
</tr>
<tr>
<td>One Mutex for Entire List</td>
<td>2.50</td>
</tr>
<tr>
<td>One Mutex per Node</td>
<td>12.00</td>
</tr>
</tbody>
</table>
Issues with Threads: False Sharing, Deadlocks, Thread-safety
Caches, Cache-Coherence, and False Sharing

- Underlying cache-memory interaction can have a significant impact on shared-memory program performance in some cases.
- Cache fetches data with a *cacheline* as a unit. Cacheline=128 bytes in Intel Xeon.
Problem: False Sharing

- Occurs when two or more processors/cores access different data in same cache line, and at least one of them writes.
  - Leads to ping-pong effect.
- Let’s assume we parallelize code with p=2:
  ```c
  for( i=0; i<n; i++ )
      a[i] = b[i];
  ```
  - Each array element takes 8 bytes
  - Cache line has 64 bytes (8 numbers)
Execute this program in two processors
for( i=0; i<n; i++ )
a[i] = b[i];

```
Execute this program in two processors
for( i=0; i<n; i++ )
a[i] = b[i];
```

```c
int a[8];
int b[8];
for( i=0; i<n; i++ )
a[i] = b[i];
```
False Sharing: Ping-Pong Effort of Cacheline Access
False Sharing: Ping-Pong Effort of Cacheline Access
False Sharing: Ping-Pong Effort of Cacheline Access
False Sharing: Ping-Pong Effort of Cacheline Access

CPU0

CPU1

L2

L2

Main memory
False Sharing: Ping-Pong Effort of Cacheline Access
False Sharing: Ping-Pong Effort of Cacheline Access
False Sharing: Ping-Pong Effort of Cacheline Access
False Sharing: Ping-Pong Effort of Cacheline Access

CPU0

CPU1

L2

L2

Main memory
False Sharing: Example

Two CPUs execute:
for( i=0; i<n; i++ )
a[i] = b[i];

- Written by CPU 0
- Written by CPU 1


Cache line

data

inv

CPU0


CPU1

Block-based pthreads matrix-vector multiplication

```c
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 */
```
**Impact of false sharing on performance of matrix-vector multiplication**

<table>
<thead>
<tr>
<th>Threads</th>
<th>Matrix Dimension</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8,000,000 × 8</td>
<td>8000 × 8000</td>
<td>8 × 8,000,000</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.393</td>
<td>1.000</td>
<td>0.345</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>0.217</td>
<td>0.906</td>
<td>0.188</td>
<td>0.918</td>
</tr>
<tr>
<td>4</td>
<td>0.139</td>
<td>0.707</td>
<td>0.115</td>
<td>0.750</td>
</tr>
</tbody>
</table>

(times are in seconds)
How to avoid false sharing?

- Avoid to write consecutive global variables from different threads
  - Use thread-specific local/private space as much as possible.
  - Pad frequently-modified global variables so they are not stored close to each other in memory and will not be held together within a cacheline.

Two CPUs execute:

```c
for( i=0; i<n; i++ )
a[i] = b[i];
```
Deadlock and Starvation

- **Deadlock** – two or more threads are waiting indefinitely for an event that can be only caused by one of these waiting threads.
- **Starvation** – indefinite blocking (in a waiting queue forever).

Let $s$ and $q$ be two mutex locks:

\[
\begin{align*}
P_0 & \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \Quad...
Deadlock Avoidance

• **Order the locks and always acquire the locks in that order.**

• **Eliminate circular waiting**

\[
\begin{align*}
P_0 & \quad P_1 \\
\text{Lock}(S); & \quad \text{Lock}(S); \\
\text{Lock}(Q); & \quad \text{Lock}(Q); \\
\cdots & \quad \cdots \\
\text{Unlock}(Q); & \quad \text{Unlock}(Q); \\
\text{Unlock}(S); & \quad \text{Unlock}(S);
\end{align*}
\]
Thread-Safety

• A block of code is **thread-safe** if it can be simultaneously executed by multiple threads without causing problems.

• When you program your own functions, you know if they are safe to be called by multiple threads or not.

• You may forget to check if system library functions used are thread-safe.
  - Unsafe function: `strtok()` from C `string.h` library
  - Other example.
    - The random number generator `random` in `stdlib.h`.
    - The time conversion function `localtime` in `time.h`.
Example of using `strtok()`

- "Tokenize" a English text file
  - Tokens are contiguous sequences of characters separated by a white-space, a tab, or a newline.
  - Example: "Take UCSB CS140"
    → Three tokens: "Take", "UCSB", "CS140"
- Divide the input file into lines of text and assign the lines to the threads in a round-robin fashion.
  - Each thread tokenizes a line using `strtok()`
  - Line 1 → thread 0, Line 2 → thread 1, . . . , the tth goes to thread t, the t +1st goes to thread 0, etc.
  - Serialize access to input lines using semaphores
The strtok function

• The first time it’s called,
  ▪ the string argument is the text to be tokenized (Our line of input)
  ▪ `strtok` caches a pointer to string
• For subsequent calls, it returns successive tokens taken from the cached copy
  ▪ the first argument should be NULL.

```c
char* strtok(
    char* string /* in/out */,
    const char* separators /* in */);
```
Multi-threaded tokenizer (1)

```c
void *Tokenize(void* rank) {
    long my_rank = (long) rank;
    int count;
    int next = (my_rank + 1) % thread_count;
    char *fg_rv;
    char my_line[MAX];
    char *my_string;

    sem_wait(&sems[my_rank]);
    fg_rv = fgets(my_line, MAX, stdin);
    sem_post(&sems[next]);
    while (fg_rv != NULL) {
        printf("Thread %ld > my line = %s", my_rank, my_line);
    }
}
```
Multi-threaded tokenizer (2)

```c
count = 0;
my_string = strtok(my_line, " \t\n");
while ( my_string != NULL ) {
    count++;
    printf("Thread %ld > string %d = %s\n", my_rank, count, my_string);
    my_string = strtok(NULL, " \t\n");
}

sem_wait(&sems[my_rank]);
fg_rv = fgets(my_line, MAX, stdin);
sem_post(&sems[next]);
}

return NULL;
} /* Tokenize */
```
Running with one thread

Input file:
Pease porridge hot.
Pease porridge cold.
Pease porridge in the pot
Nine days old.

• It correctly tokenizes the input stream with 1 thread
Pease
porridge
hot

...
Running with two threads

Thread 0 > my line = Pease porridge hot.
Thread 0 > string 1 = Pease
Thread 0 > string 2 = porridge
Thread 0 > string 3 = hot.
Thread 1 > my line = Pease porridge cold.
Thread 0 > my line = Pease porridge in the pot
Thread 0 > string 1 = Pease
Thread 0 > string 2 = porridge
Thread 0 > string 3 = in
Thread 0 > string 4 = the
Thread 0 > string 5 = pot
Thread 1 > string 1 = Pease
Thread 1 > my line = Nine days old.
Thread 1 > string 1 = Nine
Thread 1 > string 2 = days
Thread 1 > string 3 = old.
What happened?

• `strtok` caches the input line by declaring a variable to have static (persistent) storage class.
  - Unfortunately this cached string is shared, not private.
• Thus, thread 0’s call to `strtok` with the third line of the input has apparently overwritten the contents of thread 1’s call with the second line.
• So the `strtok` function is not thread-safe. If multiple threads call it simultaneously, the output may not be correct.
“re-entrant” (thread safe) functions

- In some cases, the C standard specifies an alternate, thread-safe, version of a function.

```c
char* strtok_r(
    char*  string    /* in/out */,
    const char* separators, /* in */
    char** saveptr_p    /* in/out */);
```
Concluding Remarks

- A read-write lock is used when it’s safe for multiple threads to simultaneously read a data structure while only one write thread can access the data structure during the modification.
- False sharing happens when two threads/cores frequently read/write different data items stored in the same cacheline.
- Deadlocks can happen when using thread synchronization.
- Thread-safe functions.
  - Some thread-unsafe C functions cache data between calls by declaring variables to be static, causing errors when multiple threads call the function.