Table of Contents

1. Pthreads - Controlling Access and Synchronization
   - Pthreads: Read-write Locks
     - Pthreads: One Mutex
     - Pthreads: Multiple Mutexes
     - Pthreads: Read-Write Locks
   - Pthreads: Caches
   - Pthreads: Thread Safety
   - Pacheco - Concluding Remarks

2. Next Time
Recall: we are looking at shared memory programming

Challenges:
- Multiple threads accessing (or trying to) same memory locations
- Need way to control access to shared data

Example: the shared data structure is a *sorted* linked list
- Date type: Integers
- Ops: Member, Insert, Delete
Linked Lists

```c
struct list_node_s {
    int data;
    struct list_node_s* next;
};
```
Linked List Membership

```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 */
}```
Inserting a new node into a list

```
head_p -> 2 -> 5 -> 8
          |     |   |
  pred_p  -> x  curr_p
```

```
temp_p -> 7
```
Deleting a node from a linked list

```
head_p → 2 → 5 → 8
|       |       |
|       |       |
pred_p | curr_p|
```

This diagram illustrates the process of deleting a node from a linked list. The node with the value 5 is removed, and its pointers are updated to maintain the list's integrity.
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 */
}
```
Serial Linked List Input Code

```c
/*---------------------------------------------*/
int main(void) {
    char command;
    int value;
    struct list_node_s* head_p = NULL; /* start with empty list */
    command = Get_command();
    while (command != 'q' && command != 'Q') {
        switch (command) {
        case 'i':
        case 'I':
            value = Get_value();
            Insert(value, &head_p); /* Ignore return value */
            break;
        case 'p':
        case 'P':
            Print(head_p);
            break;
        case 'm':
        case 'M':
            value = Get_value();
            Member(value, head_p); /* Ignore return value */
            break;
        case 'd':
        case 'D':
            value = Get_value();
            Delete(value, &head_p); /* Ignore return value */
            break;
        default:
            printf("There is no %c command\n", command);
            printf("Please try again\n");
            break;
        }
        command = Get_command();
    }
    Free_list(&head_p);
    return 0;
} /* main */
```
Serial Linked List Output

[mthomas] % ./ser_linked_list
Please enter a command: i
Please enter a value: 3
Please enter a command: i
Please enter a value: 5
Please enter a command: i
Please enter a value: 6
Please enter a command: i
Please enter a value: 1
Please enter a command: m
Please enter a value: 20
20 is not in the list
Please enter a command: m
Please enter a value: 35
35 is in the list
Please enter a command: p
list = 1 3 5 6 18 35
Please enter a command: d
Please enter a value: 17
17 is not in the list
Please enter a command: d
Please enter a value: 18
Please enter a command: p
list = 1 3 5 6 35
Please enter a command: q
[mthomas] %
A Multi-Threaded Linked List

- Let's try to use these functions in a Pthreads program.
- In order to share access to the list, we can 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 of Shared Memory by Two Threads: Synchronization Problem

- Th0 execs Member(5)
  - Th1 execs Delete(5)
  - Th0 may find N(5) - an error
- Th0 execs Member(8)
  - Th1 execs Delete(5)
  - If done before Th0 can follow link, N(8) won’t be found
- Insert and Delete need barrier/mutex operation
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).
Pthreads - Controlling Access and Synchronization

Pthreads: Read-write Locks

**pthreads_one_mut.c: main**

```c
int main(int argc, char* argv[]) {
    long i;
    int key, success, attempts;
    pthread_t* thread_handles;
    int inserts_in_main;
    unsigned seed = 1;
    double start, finish;
    if (argc != 2) Usage(argv[0]);
    thread_count = strtol(argv[1], NULL, 10);
    Get_input(&inserts_in_main);
    /* Try to insert inserts_in_main keys,
       but give up after */
    /* 2*inserts_in_main attempts.*/
    i = attempts = 0;
    while (i < inserts_in_main &&
           attempts < 2*inserts_in_main) {
        key = my_rand(&seed) % MAX_KEY;
        success = Insert(key);
        attempts++;
        if (success) i++;
    }
    printf("Inserted %ld keys in empty list\n", i);
    GET_TIME(start);
    for (i = 0; i < thread_count; i++)
        pthread_create(&thread_handles[i], NULL,
                        Thread_work, (void*) i);
    for (i = 0; i < thread_count; i++)
        pthread_join(thread_handles[i], NULL);
    GET_TIME(finish);
    printf("Elapsed time = %e seconds\n", finish - start);
    printf("Total ops = %d\n", total_ops);
    printf("member ops = %d\n", member_total);
    printf("insert ops = %d\n", insert_total);
    printf("delete ops = %d\n", delete_total);
    # ifdef OUTPUT
    printf("Before starting threads, list = \n");
    Print(); printf("\n");
    # endif
    thread_handles = malloc(thread_count*sizeof(pthread_t));
    pthread_mutex_init(&mutex, NULL);
    pthread_mutex_init(&count_mutex, NULL);
    GET_TIME(start);
    for (i = 0; i < thread_count; i++)
        pthread_create(&thread_handles[i], NULL,
                        Thread_work, (void*) i);
    for (i = 0; i < thread_count; i++)
        pthread_join(thread_handles[i], NULL);
    GET_TIME(finish);
    printf("Elapsed time = %e seconds\n", finish - start);
    printf("Total ops = %d\n", total_ops);
    printf("member ops = %d\n", member_total);
    printf("insert ops = %d\n", insert_total);
    printf("delete ops = %d\n", delete_total);
    # ifdef OUTPUT
    printf("After threads terminate, list = \n");
    Print(); printf("\n");
    # endif
    Free_list();
    pthread_mutex_destroy(&mutex);
    pthread_mutex_destroy(&count_mutex);
    free(thread_handles);
    return 0;
} /* main */
```
Pthreads - Controlling Access and Synchronization

Pthreads: Read-write Locks

pth.ll_one_mut.c: Thread_work: lock everything, everytime

```c
void* Thread_work(void* rank) {
    long my_rank = (long) rank;  int i, val;
    double which_op; unsigned seed = my_rank + 1;
    int my_member=0, my_insert=0, my_delete=0;
    int ops_per_thread = total_ops/thread_count;

    for (i = 0; i < ops_per_thread; i++) {
        which_op = my_drand(&seed);
        val = my_rand(&seed) % MAX_KEY;
        if (which_op < search_percent) {
            pthread_mutex_lock(&mutex);
            Member(val);
            pthread_mutex_unlock(&mutex);
            my_member++;
        } else if (which_op < (search_percent + insert_percent) ) {
            pthread_mutex_lock(&mutex);
            Insert(val);
            pthread_mutex_unlock(&mutex);
            my_insert++;
        } else { /* delete */
            pthread_mutex_lock(&mutex);
            Delete(val);
            pthread_mutex_unlock(&mutex);
            my_delete++; 
        }
    } /* for */
    pthread_mutex_lock(&count_mutex);
    member_total += my_member;
    insert_total += my_insert;
    delete_total += my_delete;
    pthread_mutex_unlock(&count_mutex);
    return NULL; }
```

```c
/---------------------*/
void Free_list(void) {
    struct list_node_s* current;
    struct list_node_s* following;

    if (Is_empty()) return;
    current = head;
    following = current->next;
    while (following != NULL) {
        # ifdef DEBUG
        printf("Freeing \%d\n", current->data);
        # endif
        free(current);
        current = following;
        following = current->next;
    }
    # ifdef DEBUG
    printf("Freeing \%d\n", current->data);
    # endif
    free(current);
} /* Free_list */

/*---------------------*/
int Is_empty(void) {
    if (head == NULL)
        return 1;
    else
        return 0;
} /* Is_empty */
```
**pth.ll_one_mut.c: INSERT and DELETE**

/* Insert value in correct numerical location into list */
/* If value is not in list, return 1, else return 0 */
int Insert(int value) {
    struct list_node_s* curr = head;
    struct list_node_s* pred = NULL;
    struct list_node_s* temp;
    int rv = 1;
    while (curr != NULL && curr->data < value) {
        pred = curr;
        curr = curr->next;
    }
    if (curr == NULL || curr->data > value) {
        temp = malloc(sizeof(struct list_node_s));
        temp->data = value;
        temp->next = curr;
        if (pred == NULL)
            head = temp;
        else
            pred->next = temp;
    } else { /* value in list */
        rv = 0;
    }
    return rv;
} /* Insert */

/* Deletes value from list */
/* If value is in list, return 1, else return 0 */
int Delete(int value) {
    struct list_node_s* curr = head;
    struct list_node_s* pred = NULL;
    int rv = 1;
    /* Find value */
    while (curr != NULL && curr->data < value) {
        pred = curr;
        curr = curr->next;
    }
    if (curr != NULL && curr->data == value) {
        if (pred == NULL) { /* first element in list */
            head = curr->next;
            # ifdef DEBUG
            printf("Freeing %d\n", value);
            # endif
            free(curr);
        } else {
            pred->next = curr->next;
            # ifdef DEBUG
            printf("Freeing %d\n", value);
            # endif
            free(curr);
        } else { /* Not in list */
            rv = 0;
        }
    return rv; } /* Delete */
Pthreads - Controlling Access and Synchronization

Pthreads: Read-write Locks

**pth.ll_one_mut.c: Utilities**

```c
void Usage(char* prog_name) {
    fprintf(stderr, "usage: %s <thread_count>
        " prog_name);
    exit(0);
} /* Usage */

/* Struct for list nodes */
struct list_node_s {
    int data;
    struct list_node_s* next;
};

int Member(int value) {
    struct list_node_s* temp;
    temp = head;
    while (temp != NULL && temp->data < value)
        temp = temp->next;
    if (temp == NULL || temp->data > value) {
        #ifdef DEBUG
            printf("%d is not in the list\n", value);
        #endif
        return 0;
    } else {
        #ifdef DEBUG
            printf("%d is in the list\n", value);
        #endif
        return 1;
    }
}
```

```c
/* Struct for list nodes */
struct list_node_s {
    int data;
    struct list_node_s* next;
};

int Member(int value) {
    struct list_node_s* temp;
    temp = head;
    while (temp != NULL && temp->data < value)
        temp = temp->next;
    if (temp == NULL || temp->data > value) {
        #ifdef DEBUG
            printf("%d is not in the list\n", value);
        #endif
        return 0;
    } else {
        #ifdef DEBUG
            printf("%d is in the list\n", value);
        #endif
        return 1;
    }
}
```

```c
/* Struct for list nodes */
struct list_node_s {
    int data;
    struct list_node_s* next;
};

int Member(int value) {
    struct list_node_s* temp;
    temp = head;
    while (temp != NULL && temp->data < value)
        temp = temp->next;
    if (temp == NULL || temp->data > value) {
        #ifdef DEBUG
            printf("%d is not in the list\n", value);
        #endif
        return 0;
    } else {
        #ifdef DEBUG
            printf("%d is in the list\n", value);
        #endif
        return 1;
    }
}
```

```c
/* Struct for list nodes */
struct list_node_s {
    int data;
    struct list_node_s* next;
};

int Member(int value) {
    struct list_node_s* temp;
    temp = head;
    while (temp != NULL && temp->data < value)
        temp = temp->next;
    if (temp == NULL || temp->data > value) {
        #ifdef DEBUG
            printf("%d is not in the list\n", value);
        #endif
        return 0;
    } else {
        #ifdef DEBUG
            printf("%d is in the list\n", value);
        #endif
        return 1;
    }
}
```
**Output from Pthread Linked List - One mutex per list node**

[mthomas@tuckoo ch4]$ ./pth_ll_one_mut 16
How many keys should be inserted in the main thread?
100
How many total ops should be executed?
100
Percent of ops that should be searches? (between 0 and 1)
.2
Percent of ops that should be inserts? (between 0 and 1)
.3
Inserted 100 keys in empty list
Elapsed time = 2.205133e-03 seconds
Total ops = 100
member ops = 18
insert ops = 32
delete ops = 46

[mthomas@tuckoo ch4]$
[mthomas@tuckoo ch4]$ ./pth_ll_one_mut 128
How many keys should be inserted in the main thread?
100000
How many total ops should be executed?
1000
Percent of ops that should be searches? (between 0 and 1)
.25
Percent of ops that should be inserts? (between 0 and 1)
.5
Inserted 100000 keys in empty list
Elapsed time = 1.773283e+00 seconds
Total ops = 1000
member ops = 220
insert ops = 454
delete ops = 222
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, then this may be the best solution since we’ll need to serialize access to the list for most of the operations, and this solution will certainly be easy to implement.
Solution #2

- Instead of locking the entire list, we could try to lock individual nodes.
- A “finer-grained” approach.

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

- This is much more complex than the original Member function.
- It is also much slower, since, in general, each time a node is accessed, a mutex must be locked and unlocked.
- The addition of a mutex field to each node will substantially increase the amount of storage needed for the list.
Implementation of Member with one mutex per list node (1)

```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;
    }
    pthread_mutex_unlock(&head_p_mutex);
    return 0;
}
```
Implementation of Member with one mutex per list node (2)

```c
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 */
```
### Output from Pthread Linked List - Multiple mutexes

```
[mthomas]% ./pth_ll_multi_mut 8
How many keys should be inserted in the main thread?
1000
How many total ops should the threads execute?
1000
Percent of ops that should be searches? (between 0 and 1)
.2
Percent of ops that should be inserts? (between 0 and 1)
.4
Inserted 1000 keys in empty list
Elapsed time = 7.796049e-03 seconds
Total ops = 1000
member ops = 205
insert ops = 403
delete ops = 392
```

```
[mthomas]% ./pth_ll_one_mut 8
How many keys should be inserted in the main thread?
1000
How many total ops should the threads execute?
1000
Percent of ops that should be searches? (between 0 and 1)
.2
Percent of ops that should be inserts? (between 0 and 1)
.4
Inserted 1000 keys in empty list
Elapsed time = 9.109974e-03 seconds
Total ops = 1000
member ops = 205
insert ops = 403
delete ops = 392
```
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.
Pthreads Read-Write Locks

- 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.
Pthreads Read-Write Locks

- 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.
Protecting our linked list functions

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

pthread_rwlock_wrlock(&rwlock);
Insert(value);
pthread_rwlock_unlock(&rwlock);

pthread_rwlock_wrlock(&rwlock);
Delete(value);
pthread_rwlock_unlock(&rwlock);
```
Pthreads - Controlling Access and Synchronization

Pthreads: Read-write Locks

```c
#define OUTPUT 0

int main(int argc, char* argv[]) {
    long i;
    int key, success, attempts;
    pthread_t* thread_handles;
    int inserts_in_main;
    unsigned seed = 1;
    double start, finish;

    if (argc != 2) Usage(argv[0]);
    thread_count = strtol(argv[1],NULL,10);
    Get_input(&inserts_in_main);

    /* Try to insert inserts_in_main keys, but
give up after 2*inserts_in_main attempts.*/
i = attempts = 0;
    while (i < inserts_in_main &&
          attempts < 2*inserts_in_main){
        key = my_rand(&seed) % MAX_KEY;
        success = Insert(key); attempts++;
        if (success) i++;
    }
    printf("Inserted %ld keys in empty list\n", i);

    thread_handles = malloc(thread_count*sizeof(pthread_t));
    pthread_mutex_init(&count_mutex, NULL);
    pthread_rwlock_init(&rwlock, NULL);
    GET_TIME(start);
    for (i = 0; i < thread_count; i++)
        pthread_create(&thread_handles[i], NULL,
                        Thread_work, (void*) i);
    for (i = 0; i < thread_count; i++)
        pthread_join(thread_handles[i], NULL);
    GET_TIME(finish);
    printf("Elapsed time = %e seconds\n", finish - start);
    printf("Total ops = %d\n", total_ops);
    printf("member ops = %d\n", member_count);
    printf("insert ops = %d\n", insert_count);
    printf("delete ops = %d\n", delete_count);

    if (OUTPUT)
        printf("Before starting threads, list = \n");
    Print();
    printf("\n");
    Free_list();

    pthread_rwlock_destroy(&rwlock);
    pthread_mutex_destroy(&count_mutex);
    free(thread_handles);
    return 0;
}
```
Pthreads: Read-write Locks

pthreads-cont-mut.c: Thread_work:

```c
void* Thread_work(void* rank) {
    long my_rank = (long) rank;
    int i, val;   double which_op;
    unsigned seed = my_rank + 1;
    int my_member_count=0, my_insert_count=0, my_delete_count=0;
    int ops_per_thread = total_ops/thread_count;
    for (i = 0; i < ops_per_thread; i++) {
        which_op = my_drand(&seed);
        val = my_rand(&seed) % MAX_KEY;
        if (which_op < search_percent) {
            pthread_rwlock_rdlock(&rwlock);
            Member(val);
            pthread_rwlock_unlock(&rwlock);
            my_member_count++;
        } else if (which_op < (search_percent +
            insert_percent)) {
            pthread_rwlock_wrlock(&rwlock);
            Insert(val);
            pthread_rwlock_unlock(&rwlock);
            my_insert_count++;
        } else { /* delete */
            pthread_rwlock_wrlock(&rwlock);
            Delete(val);
            pthread_rwlock_unlock(&rwlock);
            my_delete_count++;
        }
    } /* for */
    pthread_mutex_lock(&count_mutex);
    member_count += my_member_count;
    insert_count += my_insert_count;
    delete_count += my_delete_count;
    pthread_mutex_unlock(&count_mutex);
    return NULL; } /* Thread_work */
```

Free_list:

```c
void Free_list(void) {
    struct list_node_s* current;
    struct list_node_s* following;
    if (Is_empty()) return;
    current = head;
    following = current->next;
    while (following != NULL) {
        if (following != NULL) {
            # ifdef DEBUG
            printf("Freeing %d\n", current->data);
            # endif
            free(current);
            current = following;
            following = following->next;
        } /* for */
        printf("Freeing %d\n", current->data);
        free(current);
    } /* for */
}
```

Is_empty:

```c
int Is_empty(void) {
    if (head == NULL)
        return 1;
    else
        return 0;
} /* Is_empty */
```
/* Insert value in correct numerical location into list */
/* If value is not in list, return 1, else return 0 */
int Insert(int value) {
    struct list_node_s* curr = head;
    struct list_node_s* pred = NULL;
    struct list_node_s* temp;
    int rv = 1;

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

    if (curr == NULL || curr->data > value) {
        temp = malloc(sizeof(struct list_node_s));
        temp->data = value;
        temp->next = curr;
        if (pred == NULL) {
            head = temp;
        } else {
            pred->next = temp;
        }
    } else { /* value in list */
        rv = 0;
    }

    return rv;
} /* Insert */

/* Deletes value from list */
/* If value is in list, return 1, else return 0 */
int Delete(int value) {
    struct list_node_s* curr = head;
    struct list_node_s* pred = NULL;
    int rv = 1;

    /* Find value */
    while (curr != NULL && curr->data < value) {
        pred = curr;
        curr = curr->next;
    }

    if (curr != NULL && curr->data == value) {
        if (pred == NULL) { /* first element in list */
            head = curr->next;
        # ifdef DEBUG
            printf("Freeing %d\n", value);
        # endif
            free(curr);
        } else {
            pred->next = curr->next;
        # ifdef DEBUG
            printf("Freeing %d\n", value);
        # endif
            free(curr);
        } else { /* Not in list */
            rv = 0;
        }
    }

    return rv; } /* Delete */
void Usage(char* prog_name) {
    fprintf(stderr, "usage: %s <thread_count>\n", prog_name);
    exit(0);
} /* Usage */

void Get_input(int* inserts_in_main_p) {
    printf("How many keys should be inserted
    in the main thread?\n");
    scanf("%d", inserts_in_main_p);
    printf("How many total ops should be executed?\n");
    scanf("%d", &total_ops);
    printf("Percent of ops that should be searches?
    (between 0 and 1)\n");
    scanf("%lf", &search_percent);
    printf("Percent of ops that should be inserts?
    (between 0 and 1)\n");
    scanf("%lf", &insert_percent);
    delete_percent = 1.0 - (search_percent +
    insert_percent);
} /* Get_input */

void Print(void) {
    struct list_node_s* temp;
    printf("list = ");
    temp = head;
    while (temp != (struct list_node_s*) NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
} /* Print */

/* Struct for list nodes */
struct list_node_s {
    int data;
    struct list_node_s* next;
};

int Member(int value) {
    struct list_node_s* temp;
    temp = head;
    while (temp != NULL && temp->data < value)
        temp = temp->next;
    if (temp == NULL || temp->data > value) {
        # ifdef DEBUG
        printf("%d is not in the list\n", value);
        # endif
        return 0;
    } else {
        # ifdef DEBUG
        printf("%d is in the list\n", value);
        # endif
        return 1;
    }
} /* Member */
Linked List Performance

<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>

100,000 ops/thread
99.9% Member
0.05% Insert
0.05% Delete
## Linked List Performance

<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>

100,000 ops/thread
80% Member
10% Insert
10% Delete
Output from Pthread Linked List - Multiple mutexes

[mthomas] % ./pth_ll_rwlock 8
1000
How many ops total should be executed?
100000
Percent of ops that should be searches? (between 0 and 1) .8
Percent of ops that should be inserts? (between 0 and 1) .1
Inserted 1000 keys in empty list
Elapsed time = 1.069531e+00 seconds
Total ops = 100000
member ops = 79876
insert ops = 9964
delete ops = 10160

[mthomas] % ./pth_ll_multi_mut 8
Elapsed time = 3.434680e+00 seconds
Total ops = 100000
member ops = 79876
insert ops = 9964
delete ops = 10160

[mthomas] % ./pth_ll_one_mut 8
Elapsed time = 2.255279e+00 seconds
Total ops = 100000
member ops = 79876
insert ops = 9964
delete ops = 10160
Caches, Cache-Coherence, and False Sharing

- Recall that chip designers have added blocks of relatively fast memory to processors called cache memory.
- The use of cache memory can have a huge impact on shared-memory.
- A write-miss occurs when a core tries to update a variable that’s not in cache, and it has to access main memory.
Cache knows about spatial/temporal locality

If PE accesses data, then transfer more data. This is called cacheline or cacheblock

But, using cache impacts operations

if each thread copies a shared variable, $x$, into local cache (for performance), and then one or more threads change $x$, what is supposed to be the real value of $x$?

This is the cache coherence problem and is enforced by most systems.

look at matrix multiplication (see Ch2, and Ch4)
### Pthreads: Caches

#### Matrix Equations

\[ a_{ij} x_j = y_i \]

<table>
<thead>
<tr>
<th>( a_{i0} )</th>
<th>( a_{i1} )</th>
<th>( \cdots )</th>
<th>( a_{in-1} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a_{00} )</td>
<td>( a_{01} )</td>
<td>( \cdots )</td>
<td>( a_{0,n-1} )</td>
</tr>
<tr>
<td>( a_{10} )</td>
<td>( a_{11} )</td>
<td>( \cdots )</td>
<td>( a_{1,n-1} )</td>
</tr>
<tr>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
</tr>
<tr>
<td>( a_{i0} )</td>
<td>( a_{i1} )</td>
<td>( \cdots )</td>
<td>( a_{i,n-1} )</td>
</tr>
<tr>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
</tr>
<tr>
<td>( a_{m-1,0} )</td>
<td>( a_{m-1,1} )</td>
<td>( \cdots )</td>
<td>( a_{m-1,n-1} )</td>
</tr>
</tbody>
</table>

\[ y_i = a_{i0} x_0 + a_{i1} x_1 + \cdots + a_{i,n-1} x_{n-1} \]

<table>
<thead>
<tr>
<th>( x_0 )</th>
<th>( x_1 )</th>
<th>( \vdots )</th>
<th>( x_{n-1} )</th>
<th>( y_0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y_1 )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( y_{m-1} )</td>
</tr>
</tbody>
</table>
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 */
```
Performance

Time vs # Threads

Efficiency vs # Threads
Observations

- Performance is affected by distribution for 1 thread:
  - Write misses - minimal for 8,000,000x8 (line 9, init y)
  - Read misses - minimal for 8x8,000,000 (line 11, reading x)

- Performance improves with thread number
  - read misses: threads must reload y many times.
  - false sharing
Thread-Safety

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

- Suppose we want to use multiple threads to “tokenize” a file that consists of ordinary English text.
- The tokens are just contiguous sequences of characters separated from the rest of the text by white-space — a space, a tab, or a newline.
Simple approach

- Divide the input file into lines of text and assign the lines to the threads in a round-robin fashion.

- The first line goes to thread 0, the second goes to thread 1, ..., the tth goes to thread t, the t+1st goes to thread 0, etc.
Simple approach

- We can serialize access to the lines of input using semaphores.

- After a thread has read a single line of input, it can tokenize the line using the `strtok` function.
The `strtok` function

- The first time it’s called the string argument should be the text to be tokenized.
  - Our line of input.
- For subsequent calls, the first argument should be NULL.

```c
char* strtok(
    char* string /* in/out */,
    const char* separators /* in */);
```
The `strtok` function

- The idea is that in the first call, `strtok` caches a pointer to string, and for subsequent calls it returns successive tokens taken from the cached copy.
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)

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

- It correctly tokenizes the input stream.

Pease porridge hot.
Pease porridge cold.
Pease porridge in the pot
Nine days old.
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.

Oops!
What happened?

- **strtok** caches the input line by declaring a variable to have static storage class.
  - This causes the value stored in this variable to persist from one call to the next.

- Unfortunately for us, this cached string is shared, not private.
What happened?

- 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.
Other unsafe C library functions

- Regrettably, it’s not uncommon for C library functions to fail to be thread-safe.
- The random number generator `random` in `stdlib.h`.
- The time conversion function `localtime` in `time.h`.
“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 (1)

- A thread in shared-memory programming is analogous to a process in distributed memory programming.
- However, a thread is often lighter-weight than a full-fledged process.
- In Pthreads programs, all the threads have access to global variables, while local variables usually are private to the thread running the function.
Concluding Remarks (2)

- When indeterminacy results from multiple threads attempting to access a shared resource such as a shared variable or a shared file, at least one of the accesses is an update, and the accesses can result in an error, we have a race condition.
Concluding Remarks (3)

- A critical section is a block of code that updates a shared resource that can only be updated by one thread at a time.

- So the execution of code in a critical section should, effectively, be executed as serial code.
Concluding Remarks (4)

- **Busy-waiting** can be used to avoid conflicting access to critical sections with a flag variable and a while-loop with an empty body.
- It can be very wasteful of CPU cycles.
- It can also be unreliable if compiler optimization is turned on.
Concluding Remarks (5)

- A **mutex** can be used to avoid conflicting access to critical sections as well.
- Think of it as a lock on a critical section, since mutexes arrange for mutually exclusive access to a critical section.
Concluding Remarks (6)

- A semaphore is the third way to avoid conflicting access to critical sections.
- It is an unsigned int together with two operations: sem_wait and sem_post.
- Semaphores are more powerful than mutexes since they can be initialized to any nonnegative value.
Concluding Remarks (7)

- A **barrier** is a point in a program at which the threads block until all of the threads have reached it.

- A **read-write lock** is used when it’s safe for multiple threads to simultaneously read a data structure, but if a thread needs to modify or write to the data structure, then only that thread can access the data structure during the modification.
Concluding Remarks (8)

- Some C functions cache data between calls by declaring variables to be static, causing errors when multiple threads call the function.
- This type of function is not thread-safe.
Next Time

- Next class: 04/09/15: Begin OpenMP
- HW #5 (MPI): Due 04/09/15
- HW #6 (Pthreads & Open MP): Due 04/23/15