

CPEG 455 Lab 2

Shane Cincotta

11/11/19

Abstract:

The goals of this lab are split into 3 parts:

1. Build a linked list and implement three methods based on the linked list.
The three methods are:
 - a. *Add* which adds a key to tail of the list, and updates “tail” or potentially “head” accordingly
 - b. *Remove* which will remove and return the key pointed by “head” from the list and update “head” and “tail” properly
 - c. *Size* which will return the size of the list
2. Spawn 8 threads and have the threads implement a workload. The workload should be thread safe.
3. Design and implement a better synchronization scheme at method level in pthread for the *add* and *remove* methods. The scheme should provide the best tradeoff between maximum concurrency and lock overhead.

<pre> struct p { int v; struct p * pre; struct p * next; } </pre>	<pre> struct p * head = nullptr; struct p * tail = nullptr; void add (int v); int remove(); int size(); </pre>
---	--

Fig 1. Skeleton code for linked list

```

For(i=0; i<1K; i++){
    Add(i*thread_id);
}

For(i=0; i<100K; i++){
    Add(i);
    remove();
}

Print size().

```

Fig 2. Workload to be implemented

Detailed Strategy:

I began by making a *node* struct which contains all the above attributes needed to create the linked list.

```

struct node *head;
struct node *tail;

struct node{
    int v;
    struct node *prev;
    struct node *next;
};

```

Fig 3. Building linked list

I then began working on the three methods, *add*, *size* and *remove*.

```

int size() {
    int cnt = 0;
    struct node *current_node = head;
    while ( current_node != NULL) {
        cnt++;
        current_node = current_node->next;
    }
    return(cnt);
}

```

Fig 4. Implementing *size* method

```

4 void add(int data){
5     struct node *temp, *linkedList;
6
7     linkedList = (struct node *) malloc(sizeof(struct node));    // malloc space for LL
8     linkedList->next = NULL;
9
10    linkedList->v = data;
11
12    if (head == NULL){
13        head = linkedList;
14        tail = linkedList;
15    }
16
17    else{
18        temp = head;
19        while (temp->next != NULL)
20            temp = temp->next;
21        temp->next = linkedList;
22        tail = linkedList;    // append to the tail/end
23        linkedList->prev = temp;
24        //printf("%d", tail);
25    }
26 }

```

Fig 5. Implementing *add* method

```

//remove is a reserved word
int remove1(){
    if (head == NULL){
        return NULL;
    }

    // Move the head pointer to the next node
    struct node* temp = head;
    head = head->next;

    free(temp);

    return head;
}

```

Fig 6. Implementing *remove* method

While implementing the remove method, I had to use the name *remove1* because *remove* is a reserved word from the *stdio* library.

In addition to the preceding methods, I also created two more helper methods to aid in debugging, *print* and *freeList*. *Print* prints the contents of the list and *freeList* clears the linked list.

```

void print (struct node *head) {
    struct node *current_node = head;
    while ( current_node != NULL) {
        printf("%d ", current_node->v);
        current_node = current_node->next;
    }
}

```

Fig 7. Implementing *print* method

```

void freeList(){
    if (head == NULL){
        return NULL;
    }

    while(head!= NULL){
        struct node* temp = head;
        head = head->next;

        free(temp);
    }
}

```

Fig 8. Implementing *freeList* method

I then implemented the *workload* method and wrapped it in a lock to make it thread safe.

```

void workload(){
    pthread_mutex_lock(&lock1);
    counter++;

    printf("\n Job %d has started\n", counter);

    for(int i=0; i<1000; i++){
        add(i *tid);
    }

    for(int i=0; i<1000; i++){
        add(i);
        remove1();
    }

    printf("%d\n", size());
    printf("\n Job %d has finished\n", counter);

    pthread_mutex_unlock(&lock1);
}

```

Fig 9. Implementing *workload* with lock

Once I had my core methods, I began developing a testbench within *main* for my workload. I used the *gettimeofday* method from the *time* library to calculate the execution time of the workload. 8 threads were generated using the *pthread* library and ran on the workload. A singular lock was wrapped around the workload to make it thread safe. In addition, the workload was ran 50 times and the execution times were averaged. After each iteration, the *freeList* method was used to clear the linked list to keep the size consistent.

```
int main(int argc, char *argv[]){
    int iterations = 0;
    int max_iterations = 50;
    struct timeval t1, t2;
    double elapsedTime;
    double total_elapsedTime = 0;

    int i = 0;
    while(iterations < max_iterations){
        if (pthread_mutex_init(&lock1, NULL) != 0) {
            printf("\n mutex init has failed on lock1\n");
            return 1;
        }

        gettimeofday(&t1, NULL);
        // Let us create 8 threads
        for (i = 0; i < 8; i++) {
            pthread_create(&tid, NULL, workload, NULL);
            pthread_join(tid, NULL);
        }

        gettimeofday(&t2, NULL);

        // Calculating total time taken by the program in ms
        elapsedTime = (t2.tv_sec - t1.tv_sec) * 1000.0;
        elapsedTime += (t2.tv_usec - t1.tv_usec) / 1000.0;
        total_elapsedTime += elapsedTime;

        freeList();
        iterations++;
    }

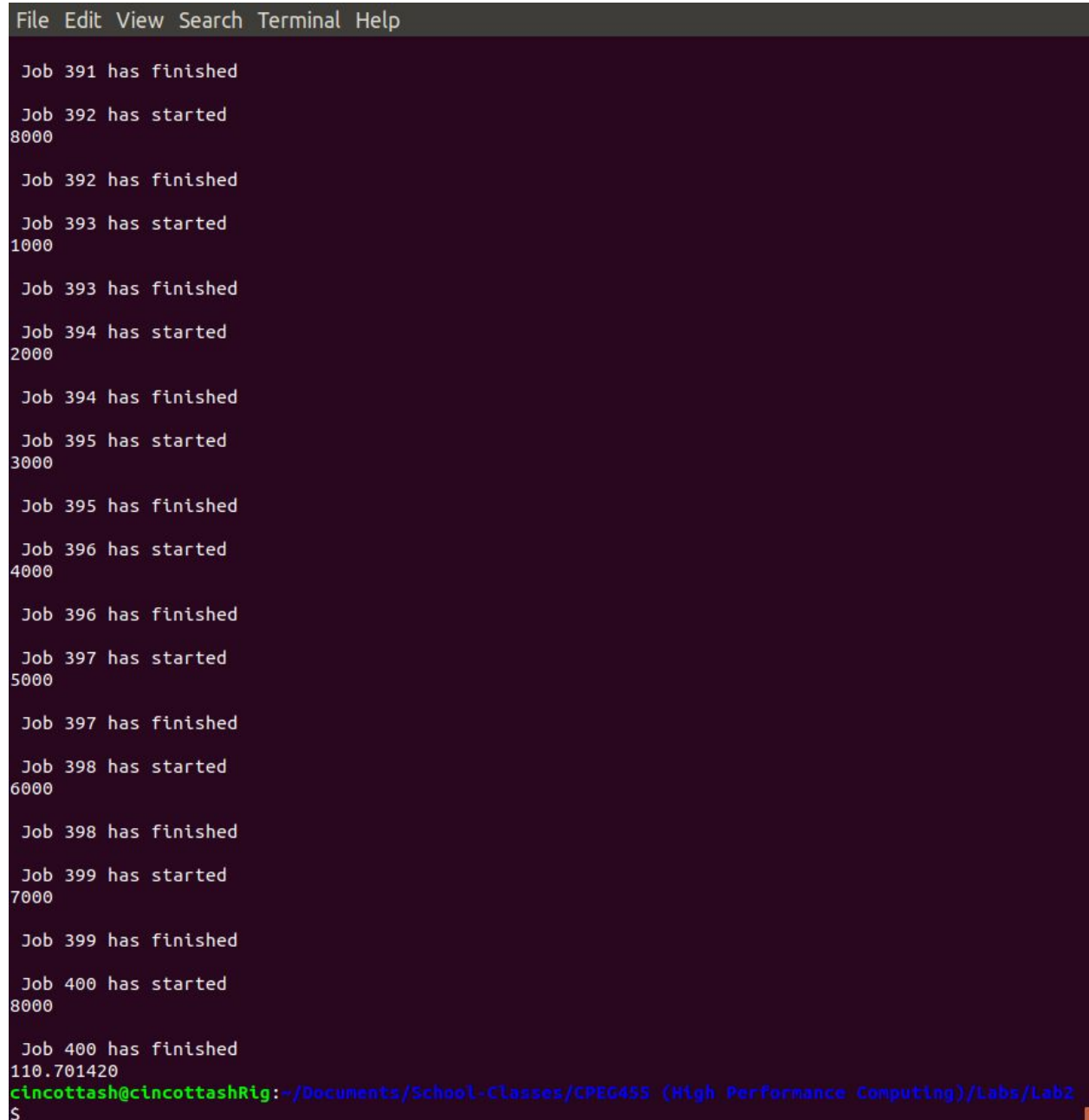
    pthread_mutex_destroy(&lock1);
    printf("%f\n", total_elapsedTime/max_iterations);
    pthread_exit(NULL);

    return 0;
}
```

Fig 9. Implementation of *main* method

Results:

The code created so far will serve as my baseline/unoptimized program for which I will test my optimized version against.



```
File Edit View Search Terminal Help

Job 391 has finished
Job 392 has started
8000
Job 392 has finished
Job 393 has started
1000
Job 393 has finished
Job 394 has started
2000
Job 394 has finished
Job 395 has started
3000
Job 395 has finished
Job 396 has started
4000
Job 396 has finished
Job 397 has started
5000
Job 397 has finished
Job 398 has started
6000
Job 398 has finished
Job 399 has started
7000
Job 399 has finished
Job 400 has started
8000
Job 400 has finished
110.701420
cincottash@cincottashRig:~/Documents/School-Classes/CPEG455 (High Performance Computing)/Labs/Lab2
$
```

Fig 10. Results of executing unoptimized version

From fig 10, the average execution time after 50 iterations was 110.7 ms. It is also noteworthy that the size of the list is reset after each iteration using *freeList*, this is to maintain a constant list size (8000).

To create an optimized version, I removed my lock around the entire workload and instead used two locks in two critical sections. I determined the critical sections by taking advantage of the way *add* and *remove1* operate. *Add* will append a key to the list and *remove1* will remove and return the head. *Add* and *remove1* can always run concurrently because they operate on different parts of the list, thus they are not a critical section. If two invocations of *add* or *remove1* are called simultaneously, a critical section is created because both function calls will be accessing the same list address. To avoid this possible race condition I used two locks, one around each of the calls to *add*. This is to avoid having multiple threads try and call *add*, which would create a possible race conditions. I avoided using more locks because using more than 2 at a time led to a decrease in performance. This is likely due to the fact that using locks is an expensive and lock overhead is quickly reached.

```

105 //
106 void workload(){
107
108     counter++;
109
110
111     printf("\n Job %d has started\n", counter);
112
113     for(int i=0; i<1000; i++){
114         pthread_mutex_lock(&lock1);
115         add(i *tid);
116         pthread_mutex_unlock(&lock1);
117     }
118
119
120     for(int i=0; i<1000; i++){
121         pthread_mutex_lock(&lock2);
122         add(i);
123         pthread_mutex_unlock(&lock2);
124
125         remove1();
126
127     }
128
129
130     printf("%d\n", size());
131
132     printf("\n Job %d has finished\n", counter);
133     //pthread_mutex_unlock(&lock1);
134
135 }
136

```

Fig 11. Optimized workload


```
File Edit View Search Terminal Help
Job 393 has finished
Job 394 has started
2000
Job 394 has finished
Job 395 has started
3000
Job 395 has finished
Job 396 has started
4000
Job 396 has finished
Job 397 has started
5000
Job 397 has finished
Job 398 has started
6000
Job 398 has finished
Job 399 has started
7000
Job 399 has finished
Job 400 has started
8000
Job 400 has finished
103.077780
cincottash@cincottashRig:~/Documents/School-Classes/CPEG455 (High Performance Co
mputing)/Labs/Lab2$
```

Fig 12. Results of executing optimized version

My optimized version resulted in a decrease of 7.7 ms.

Conclusion:

The results of the experiment were overall a success. I was able to build a testing environment for two different versions of synchronous. The placement and number of locks led to a decrease in execution time. However, using more than 2 locks led to a decrease in performance.

The source code for this lab is included in the tar file. There are 2 programs to be ran, *optimized.c* and *unoptimized.c*. To compile and run, I used the following commands, `gcc optimized.c -o optimized -Wall -lpthread` and `gcc unoptimized.c -o unoptimized -Wall -lpthread`.