Operating Systems Lab #3

David Tyler

11/13/14

Objective

Learn how to do memory management in a multithreaded environment.

Background

Malloc is a function that allocates a block of memory of the requested size and returns a pointer to the first address in it. Memory management can be done by keeping track of allocated memory - both the size and the start of each block - and making sure newly allocated memory does not overlap with previously assigned memory. Various strategies such as First Fit (fitting new memory into the first available free space in memory), Best Fit (fitting new memory into the smallest possible free block), and Worst Fit (fitting new memory into the largest possible free block) can be used to try to minimize the memory fragmentation. However, the best strategy space-wise is simple to compact the memory so that free memory is always in one large block at the end.

Algorithm

The algorithm for this lab was very simple. Each thread would wake up, try to free the memory from last time it was awake, then request a new block of memory of a random size. The thread would then go back to sleep for a random amount of time. To allocate memory, the memory_allocate() function iterates through the array of currently allocated memory until it finds the last allocated item. It then adds a new item on to the end with a pointer into memory based on the total size of the previously allocated memory in bytes. This is done by treating the malloc'ed memory as a character array and using the total size of allocated memory as an index into this array and getting the memory location of this. The size of the allocated memory and the number of the thread are also saved. The memory_free() function loops through the allocated memory until it finds the element with the thread id that was passed into the function. It then overwrites this element with the next element in the array and does this for each element until the end of the array. Each time an element is moved, its pointer into allocated memory is recalculated to account for the fact the thread id element was removed. This is compaction and leaves all the free memory at the end of the allocated memory. This also makes allocating new items easy since they can simply be added to the end.

Results

```
# ./lab3.exe 5
Creating thread: 0
Creating thread: 2
Creating thread: 1
Creating thread: 4
Creating thread: 3
I am thread 0 and I am waking up
```

```
Size requested: 171
====BEGIN MALLOC====
malloc size 171
====END MALLOC====
Current Memory: |Thread 0 @ -1995214849 (171 bytes) |
TOTAL SIZE: 171
Location: -1995214849
I am thread 0 and I am going to sleep
I am thread 2 and I am waking up
Size requested: 237
====BEGIN MALLOC====
malloc size 237
====END MALLOC====
Current Memory: |Thread 0 @ -1995214849 (171 bytes) |Thread 2 @ -1995214678 (237 bytes) |
TOTAL SIZE: 408
Location: -1995214678
I am thread 2 and I am going to sleep
I am thread 1 and I am waking up
Size requested: 314
====BEGIN MALLOC====
malloc size 314
====END MALLOC====
Current Memory: |Thread 0 @ -1995214849 (171 bytes) |Thread 2 @ -1995214678 (237 bytes) |Thread 1 @
-1995214441 (314 bytes) |
TOTAL SIZE: 722
Location: -1995214441
I am thread 1 and I am going to sleep
I am thread 4 and I am waking up
Size requested: 676
====BEGIN MALLOC====
malloc size 676
====END MALLOC====
Current Memory: |Thread 0 @ -1995214849 (171 bytes) |Thread 2 @ -1995214678 (237 bytes) |Thread 1 @
-1995214441 (314 bytes) |Thread 4 @ -1995214127 (676 bytes) |
TOTAL SIZE: 1398
Location: -1995214127
I am thread 4 and I am going to sleep
I am thread 3 and I am waking up
Size requested: 92
====BEGIN MALLOC====
malloc size 92
====END MALLOC====
Current Memory: |Thread 0 @ -1995214849 (171 bytes) |Thread 2 @ -1995214678 (237 bytes) |Thread 1 @
-1995214441 (314 bytes) |Thread 4 @ -1995214127 (676 bytes) |Thread 3 @ -1995213451 (92 bytes) |
TOTAL SIZE: 1490
Location: -1995213451
I am thread 3 and I am going to sleep
I am thread 1 and I am waking up
```

```
****BEGIN FREE****
Freeing thread 1's memory
****END FREE****
Current Memory: |Thread 0 @ -1995214849 (171 bytes) |Thread 2 @ -1995214678 (237 bytes) |Thread 4 @
-1995214441 (676 bytes) |Thread 3 @ -1995213765 (92 bytes) |
TOTAL SIZE: 1176
Size requested: 797
====BEGIN MALLOC====
malloc size 797
====END MALLOC====
Current Memory: |Thread 0 @ -1995214849 (171 bytes) |Thread 2 @ -1995214678 (237 bytes) |Thread 4 @
-1995214441 (676 bytes) |Thread 3 @ -1995213765 (92 bytes) |Thread 1 @ -1995213673 (797 bytes) |
TOTAL SIZE: 1973
Location: -1995213673
I am thread 1 and I am going to sleep
```

Conclusions

There were no problems with this lab. I did not implement the first, best, worst fit functions because I was told that we didn't need to if we did compaction.

Readme

```
OS Lab 3
David Tyler

This lab is about memory management. Memory management is critical to
making sure that memory is properly guarded against overlap from
other programs as well as to make sure that excessive fragmentation
does not occur. Compaction is used in this program to greatly
simplify allocating of memory. malloc() is used to initially create a
block of memory.

My source code file is main.c

Build the project by typing:
make

Then, run the project using:
./lab3.exe [number of threads]
```

Source

```
/*******************************************************************
 * Operating Systems
 * Program 3
 *
 * By: David Tyler
 *
 * Memory management in a multithreaded environment
 *
 * I basically keep track of the allocated memory in m_map[][]
 * ****************************************************************/



#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>

#define MAX_SIZE 4096 //Amount of RAM used for this simulation
#define MAX_BLOCK 1024 //Max amount allowed to be requested by thread
#define MAX_POOL 200 //max number of user threads

//Array to hold the thread pointers
pthread_t thread_pool[MAX_POOL];

//Array to keep track of allocated memory
int m_map[MAX_POOL][3] = {}; //pointer,size,thread

//Function Prototypes
int memory_malloc(int size, int thread_id);
int memory_free(int thread_id);
void *User(void *arg);
void display_mem(void);

//pointer to start of malloc'ed space
char *head_p;

//Mutexes
pthread_mutex_t mutx;

int main(int argc, char **argv) {
    int threads;
    int i;

    pthread_mutex_init(&mutx,NULL);

    //Malloc MAX_SIZE bytes and treat it as a character array from now on (byte array)
    head_p = (char *)malloc(MAX_SIZE);

    if(argc != 2)
    {
        printf("Usage: lab3 [num threads]\n");
        exit(1);
    }
    threads = atoi(argv[1]);
    if(threads>MAX_POOL)
    {
        printf("Number of threads must be %d or less!\n",MAX_POOL);
        exit(1);
    }
    for(i=0;i<threads;i++)
    {
        pthread_create(&thread_pool[i],NULL,User,(int *)i);
    }
    //Wait for all threads to exit
    for(i=0;i<threads;i++)
    {
        pthread_join(thread_pool[i], NULL);
    }

    return 0;
}

void *User(void *arg) {
    int size,delay;
    int location;
    //build constant delay time
    delay = (arc4random() % 9)+1; //Sleep between 1 and 10 sec
    printf("Creating thread: %d\n", arg);
    size = (arc4random() % MAX_BLOCK); //decide how much memory to request
    for(;;)
    {
        pthread_mutex_lock(&mutx);
        printf("I am thread %d and I am waking up\n",arg);
        if(location) {
            memory_free((int)arg);
        }
        size = (arc4random() % MAX_BLOCK); //decide how much memory to request
        printf("Size requested: %d\n",size);
        location = memory_malloc(size,(int)arg);
        printf("Location: %d\n",location);
        printf("I am thread %d and I am going to sleep\n",arg);
        pthread_mutex_unlock(&mutx);
        sleep(delay);
    }
}

//Iterate through m_map and look for first open element. Since the array is always compacted,
//this will also always be the last. Take the sum of the sizes of the previous elements and
use that
//to compute the pointer in the malloc'ed memory
int memory_malloc(int size, int thread_id)
{
    int mem_inuse = 0;
    int i;
    printf("====BEGIN MALLOC====\n");
    for(i=0;i<MAX_POOL;i++)
    {
        if(!m_map[i][0])
        {
            printf("malloc size %d\n",size);
            if(mem_inuse+size>MAX_SIZE)
            {
                //Memory is full, take memory from the current last element in the array
                printf("!!===> Memory full! size= %d <===!!\n Seizing memory from thread
%d\n",mem_inuse,m_map[i-1][2]);
```

```
                memory_free(m_map[i-1][2]);
                return memory_malloc(size,thread_id); //Tail call recursion
            }
            //We found the last element of the array, insert our new memory pointer,size, and
thread id here
            m_map[i][0] = (int)&head_p[mem_inuse-1]; //compute mem pointer by indexing on
total memory already in use
                                                     //into malloc space as char pointer
            m_map[i][1] = size;
            m_map[i][2] = thread_id;
            break;
        }
        mem_inuse += m_map[i][1]; //increment total memory used
    }
    printf("====END MALLOC====\n");
    display_mem(); //print out memory usage
    return m_map[i][0];
}

//Iterate though m_map and find the thread_id. Assign the next element in m_map to the place
that was occupied by thread_id.
//Iterate though the remainder of the array moving the next element to the current element and
recalulating the memory
//pointers in order to compact the memory
int memory_free(int thread_id)
{
    int i;
    int size=0;

    printf("****BEGIN FREE****\n");
    printf("Freeing thread %d's memory\n",thread_id);
    //Find thread_id
    for(i=0;i<MAX_POOL;i++)
    {
        if(m_map[i][2]==thread_id)
            break;
        size+=(int)m_map[i][1];
    }
    //Shift elements to the left in the array after removing thread_id's element
    for(;i<MAX_POOL;i++)
    {
        if(!m_map[i+1][0])
        {
            m_map[i][0] = 0;
            m_map[i][1] = 0;
            break;
        }
        m_map[i][0]=(int)&head_p[size-1]; //recompute pointer to memory for each element after
removed one
        m_map[i][1]=m_map[i+1][1];
        m_map[i][2]=m_map[i+1][2];
        size+=m_map[i][1];
    }
    printf("****END FREE****\n");
    display_mem();
    return 0;
}
```

```c
void display_mem(void)
{
    int i;
    int size=0;

    printf("Current Memory: |");
    for(i=0;i<MAX_POOL;i++)
    {
        if(m_map[i][0]==0)
        {
            printf("\nTOTAL SIZE: %d\n",size);
            return;
        }
        printf("Thread %d @ %d (%d bytes) |",m_map[i][2],m_map[i][0],m_map[i][1]);
        size+=m_map[i][1];
    }
}
```