Pablo Ruiz

pablo\_ruiz@student.uml.edu

Lab 3 report

Due 11/22/2021

Objectives:

The purpose of this lab is to write a program which will create a number of threads, N, passed by the user as the first argument. Each of these threads will request a random amount of memory from the memory management thread which will then return the address to the first element in the allocated space for that thread. Additionally, the memory management system should be able to use three different storage strategies (first fit, best fit, worst fit) as indicated by the seconds command line argument passed by the user. Ultimately, the program should also be able to use fragmentation in order to increase the amount of usable space within the allocated memory by rearranging and joining smaller empty blocks together into one large empty block.

Background:

The memory in a system is allocated by the OS upon request by an application. The memory management system (MMS) within the OS is responsible for keeping track of what memory is allocated and what is free at any given time. Additionally, it should also manage the memory as efficiently as possible in order to prevent small blocks of memory from being isolated and becoming too small to hold anything useful. The process through which the MMS separates memory empty memory blocks from allocated ones and then combines the smaller blocks into one large block is called fragmentation.

Algorithms:

The functionality of this program has been segmented into multiple different functions with each fulfilling a specific role. The main function, where it all begins, is responsible for generating the threads that will request memory and the memory management thread as well as waiting for all of them to exit so that the parent process can exit as well. In this case the function that each thread is initialized with is called “thread” and takes the rank of each thread as an argument. The function that the memory management thread is initialized with is named “mms”, this function does not receive any arguments.

The memory management function is responsible for allocating the initial block of memory from the system and initializing it in the “memory” list which will hold all of the memory blocks, allocated or empty. Each block of memory is represented as an instance of a class which includes the address of the first element in the memory block, the size of the memory block (number of bytes), and a flag representing if the block is allocated or free. Finally the memory management thread will unlock the mutex for the threads and allow them to begin queuing up and requesting memory. Subsequently the memory management thread will remain active until all other threads have exited at which point it will also exit.

Each thread function waits until the mutex is unlocked at which point it requests a random number of bytes from the “memory\_malloc” function. If it receives a NULL address it rejoins the queue and continues requesting memory until it receives a valid address. After being allocated memory it unlocks the mutex to allow other threads to request memory and sleeps for a random amount of seconds between 0 and 10. After it wakes up it waits for the mutex again and then releases its memory by calling the “memory\_free” function. Finally, it exits.

The memory\_malloc function receives a request for a number of bytes from a thread and returns an address if the request is smaller than the memory currently available. It uses the second option passed by the user when the program was called in order to select which selection strategy to use: first fit, best fit, or worst fit.

The memory free function receives an address from a thread and finds the corresponding memory block instance matching that address. Once, it has been found it changes its flag to free signifying that the memory is now free to use by other threads. Additionally, it checks if any adjacent memory blocks with continuous addresses are also free and if so merges them together to create a larger block by using the consolidate function. Finally, if the user passed the fragmentation option when the program was called it will call the fragmentation function and iterate over all of the blocks, grouping all the free blocks together and ultimately merging them to create one large free block.

The ”first\_fit”, “best\_fit”, and “worst\_fit” functions all work very similarly. They iterate over all of the memory blocks to find the one that offers the best match according to that functions criteria. The first fit approach selects the first blocks of memory that can fit the requested number of bytes. The best fit approach selects the memory block which is larger but closest in size to the requested number of bytes. Ultimately, the worst fit approach selects the block which is larger but furthest in size from the amount of bytes requested. Once they have found the correct block they all call the “split” function to separate the block into two if it is of larger size than the number of bytes requested. The original block will be cut down to the size requested and any left over bytes will be made into a new block so they can be allocated to other threads.

Results:

**Input:**



**Output:**

Text

Description automatically generated

Observations:

The use of a page table is critical when allocating memory and using dynamic fragmentation. Once and block of memory has been allocated to a process it is very hard to directly change that address without potentially affecting that process. Therefore, when allocating memory dynamically as is the case when using fragmentation, it is imperative to have a table containing logical addresses that correspond to physical spaces in memory. This way you can allocated non-consecutive physical addresses in a single block and change them as necessary without changing the address that was passed to the process.

Conclusions:

In conclusion, there are tradeoffs that must be made in memory management. Although best fit and fragmentation strategies clearly allocate the memory more efficiently. they also require additional computation time and thus require more CPU cycles which could otherwise be dedicated to other processes. Additionally, adding a layer of abstraction between the processes and the physical memory addresses allows the memory management system more flexibility when allocating memory dynamically.

Source Code:

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*                          Programmer: Pablo Ruiz                             \*/

/\*                                                                             \*/

/\*                          Course: Operating Systems                          \*/

/\*                                                                             \*/

/\*                          Project: Assignment 3                              \*/

/\*                                                                             \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

#include <mutex>

#include <string>

#include <vector>

#include <pthread.h>

#include <unistd.h>

#include <time.h>

#define MAX\_SIZE 150

using namespace std ;

class gap{

    public:

    char \*addr ;

    int size ;

    bool state ; // True = Ocuppied, False = Free

    gap(char \*start, int length, bool occupied){

        addr = start ;

        size = length ;

        state = occupied ;

    }

    void set(char \*start, int length, bool occupied){

        addr = start ;

        size = length ;

        state = occupied ;

    }

};

// Global variables

void \*mem ;

int activeUsers = 0 ;

string option1 ; // first\_fit, best\_fit, or worst\_fit

string option2 ; // frag or no\_frag

vector<gap> memory ;

mutex manager,

      user;

// Checks to see if the adjacent gaps have continuous addresses and are free, if the are they are merged back together into one gap

void consolidate(vector<gap>::iterator itr){

    vector<gap>::iterator next,

                          prev ;

    if (itr != memory.end()-1){

        next = itr + 1 ;

        if (next->state == false && next->addr == itr->addr + itr->size){

            itr->size += next->size ;

            memory.erase(next) ;

        }

    }

    if (itr != memory.begin()){

        prev = itr - 1 ;

        if (prev->state == false && prev->addr == itr->addr - prev->size){

            prev->size += itr->size ;

            memory.erase(itr) ;

        }

    }

}

// Defragments all of the empty gaps to the back and all of the full memory to the front

void defrag(){

    int index ;

    vector<gap>::iterator j ;

    // separate free gaps from full gaps

    for (vector<gap>::iterator i = memory.begin(); i < memory.end(); i++){

        if (i->state == false && i != memory.end()-1) {

            if ((i+1)->state == true){

                index = i - memory.begin() ;

                swap(memory[index], memory[index+1]) ;

            }

        }

    }

    // merge empty gaps

    for (vector<gap>::iterator i = memory.begin(); i < memory.end(); i++){

        if (i->state == false){

            j = i + 1 ;

            while (j < memory.end()){

                i->size += j->size ;

                memory.erase(j) ;

                j = i + 1 ;

            }

            break ;

        }

    }

}

void \*split(vector<gap>::iterator itr, int numBytes){

    void \*allocated = itr->addr ;

    int diff = itr->size - numBytes ;

    if (numBytes < itr->size){

        gap leftover (itr->addr + numBytes, diff, false) ;

        gap newItr (itr->addr, numBytes, true) ;

        memory.erase(itr) ;

        memory.insert(memory.begin(), leftover) ;

        memory.insert(memory.begin(), newItr) ;

    }

    else if (numBytes == itr->size){

        itr->state = true ;

    }

    return allocated ;

}

// First fit memory selection

void \*first\_fit(int numBytes){

    void \*allocated = NULL ;

    // Check for a gap that fits the required memory

    for (vector<gap>::iterator i = memory.begin(); i < memory.end(); i++){

        if (i->size >= numBytes && i->state == false){

            allocated = split(i, numBytes) ;

            break ;

        }

    }

    return allocated ;

}

// Best fit memory selection

void \*best\_fit(int numBytes){

    void \*allocated = NULL ;

    int bestDelta = MAX\_SIZE,

        currentDelta = 0 ;

    vector<gap>::iterator bestIt = memory.end() ;

    for (vector<gap>::iterator i = memory.begin(); i < memory.end(); i++) {

        currentDelta = i->size - numBytes ;

        if (bestDelta > currentDelta && currentDelta >= 0 && i->state == false) {

            bestDelta = currentDelta ;

            bestIt = i ;

        }

    }

    if (bestIt == memory.end())

        return NULL ;

    else

        allocated = split(bestIt, numBytes) ;

    return allocated ;

}

// Worst fit memory selection

void \*worst\_fit(int numBytes){

    void \*allocated = NULL ;

    int worstDelta = -MAX\_SIZE,

        currentDelta = 0 ;

    vector<gap>::iterator bestIt = memory.end() ;

    for (vector<gap>::iterator i = memory.begin(); i < memory.end(); i++) {

        currentDelta = i->size - numBytes ;

        if (worstDelta < currentDelta && currentDelta >= 0 && i->state == false) {

            worstDelta = currentDelta ;

            bestIt = i ;

        }

    }

    if (bestIt == memory.end())

        return NULL ;

    else

        allocated = split(bestIt, numBytes) ;

    return allocated ;

}

// Selects a type of memory allocation. If an incorrect option is passed it uses best fit

void \*memory\_malloc(int numBytes){

    void \*addr = NULL ;

    if (numBytes > MAX\_SIZE){

        cout << "Memory request larger than MAX\_SIZE, rejoining the queue and retrying with lower value." << endl ;

        return addr ;

    }

    if (option1 == "first\_fit") {

        addr = first\_fit(numBytes) ;

    }

    else if (option1 == "worst\_fit")

        addr = worst\_fit(numBytes) ;

    else

        addr = best\_fit(numBytes) ;

    return addr ;

}

// Frees the gaps corresponding to the passed address/es and tries to join them with adjacent gaps if there addresses are continuous and they are free, can also perform defragmentation

void memory\_free(void \*addr){

    for (vector<gap>::iterator i = memory.begin(); i < memory.end(); i++) {

        if (addr == i->addr && i->state == true) {

            i->state = false ;

            consolidate(i) ;

            if (option2 == "frag")

                defrag() ;

        }

    }

}

// Threads requesting memory

void \*thread(void \*rank){

    void \*addr = NULL ;

    int requestSize,

        sleepTime ;

    while (addr == NULL) {

        requestSize = rand() % (int)(1.3 \* MAX\_SIZE) + 1 ;

        sleepTime = rand() % 10 ;

        // Sleep to allow other threads to release their memory

        sleep(1) ;

        // Wait until the user mutex is unlocked

        user.lock() ;

        cout << endl << "I am thread " << \*(int \*)rank << ": requesting " << requestSize <<" bytes and going to sleep for " << sleepTime << " seconds." << endl ;

        addr = memory\_malloc(requestSize) ;

        // Unlock the mutex so that other threads can access the memory

        user.unlock() ;

    }

    // Sleep for a maximum of 10 sec

    sleep(sleepTime) ;

    user.lock() ;

    cout << endl << "I am thread " << \*(int \*)rank << " waking up" << endl ;

    memory\_free(addr) ;

    cout << "I am thread " << \*(int \*)rank << ": " << requestSize << " bytes released succesfully. Exiting now." << endl ;

    user.unlock() ;

    activeUsers-- ;

}

// Memory management system which allocates memory from system and coordinates the requesting threads

void \*mms(void \*data){

    // Lock the users from interacting with memory until it has been allocated

    user.lock() ;

    // Allocate memory to be managed from the system

    gap mem ((char\*)malloc(MAX\_SIZE), (int)MAX\_SIZE, false) ;

    // Add the empty memory space to the list

    memory.push\_back(mem) ;

    // Wait until the manager mutex is unlocked

    manager.lock() ;

    // Unlock the users

    user.unlock() ;

    while (activeUsers > 0) {}

    manager.unlock() ;

}

int main( int argc, char \*argv[] ){

    int numThreads = atoi(argv[1]),

        threadRank[numThreads],

        i ;

    option1 = argv[2],

    option2 = argv[3] ;

    // Seed random function

    srand(time(0)) ;

    // Initialize thread ID array to store PIDs of every thread

    pthread\_t threadID[numThreads+1] ;

    // Lock manager mutex so that we can control when it starts

    manager.lock() ;

    // Initialize Memory Management Thread

    pthread\_create(&threadID[0], NULL, mms, NULL) ;

    // Loop once for every provider rquired

    for (i = 0; i < numThreads; i++) {

        // Store the providers rank so a pointer to it can be passed to the thread

        threadRank[i] = i + 1 ;

        // Create a provider thread and pass it its rank

        pthread\_create(&threadID[i+1], NULL, thread, &threadRank[i]) ;

        // Increase the number of active users

        activeUsers++ ;

    }

    // Unlock the manager allowing it to begin

    manager.unlock() ;

    // Wait for all threads to exit

    for (i = 0; i < numThreads+1; i++)

        pthread\_join(threadID[i], NULL) ;

    return 0 ;

}