CS 416: Assignment 2 (Flowback)

Andrew Khazanovich (akhaz)
Pranav Katkamwar (pranavk)
Karl Xu (kx33)

May 1, 2017

1 Introduction

1.1 Overview

Our memory manager follows the specified requirements for the assignment. It is integrated with our scheduler, which makes memory calls (malloc/free) directly by calling myallocate and mydeallocate, passing LIBRARYREQ as an argument to distinguish scheduler memory calls from memory calls by the threads in a user program. Calls to malloc and free from user program threads are converted by a macro to calls to myallocate and mydeallocate with THREADREQ as an argument.

1.2 Memory Manager

Our memory manager allocates memory from 8MB (1000 pages comprising the OS region, 1048 pages comprising the user region) main memory and 16MB swap file, presenting the illusion of contiguous memory to all threads. This is accomplished by allocating memory requests from all threads starting from page 1000 in the 8MB main memory (1st page in the user region). Thus, every thread thinks that its allocated memory begins at the same place: the beginning of the user region.

In order to allow for this scheme, each initial memory request must find a free page in main memory or the swap file so that page 1 in the user region can be given to the currently running thread. Subsequent memory requests will be served similarly, with each additional page being swapped into the next adjacent space to the thread's last page.

1.3 Scheduler

Our scheduler uses a multi-level priority queue consisting of 5 priority levels, with 0 being the highest priority level and 4 being the lowest priority level. The time slice given to the thread selected to run is calculated by: $timeslice = (prioritylevel + 1) * 50ms$. Our scheduler also features a maintenance cycle to prevent starvation of threads. Every 20 scheduler cycles, the aging algorithm checks the lifetime of any threads in the MLPQ, and any thread whose liftime exceeds 1000 ms is added to the highest priority level.

The main integration area with the memory manager is the state of the shared global variables current_tid and prev_tid. They hold the thread id of the currently running thread and the previously running thread, respectively, allowing the memory manager access to this information when allocating memory, swapping pages, and protecting/unprotecting pages. These variables are updated in the scheduler as needed upon context switches.

2 Important Elements

2.1 Files

The files needed to build this program are:

myallocate.h
myallocate.c
paging.h
paging.c
my_pthread_t.h
my_pthread.c

2.2 Helper Functions

void* getHead(int req, int flag)

This helper function is called by myallocate to get a pointer to the 1st page owned by the currently running thread. If the thread does not have any pages yet, the getHead function attempts to find a free page in main memory, which is swapped with the 1st page in the user region, giving the thread access to the beginning of main memory. This presents to every thread the illusion of contiguous memory starting from the 1st page in the user region, allowing each thread to believe it has access to the entire memory space. If a free page is not found in main memory, the getHead function attempts to find a free page in the swap file. If this is unsuccessful, there are no free pages left, and the function returns NULL.

This function is also called by mydeallocate to get a pointer to the 1st page owned by the currently running thread. If the thread does not have any pages yet, then it made an invalid mydeallocate call, and the function prints an error message and exits.

void* requestPage()

This helper function is called by myallocate to get additional free pages when it runs out of space in the pages it already owns. It is called within a loop that allows the currently running thread to obtain as many pages as it needs to satisfy a memory request, provided that there is enough remaining space.

void initMem(int req, void* newHead)

This helper function initializes the malloc metadata for the 1st page given to a thread when it requests memory for the 1st time. Any additional memory requests can then be handled by building off of the initial metadata.

void organizeMem(Meta* curr, int size)

This helper function is called when a memory request can be served from a free block of sufficient size to chunk off another metadata component and free memory block. It takes a pointer to a piece of memory and allocates only a block of size necessary only to fulfill the memory request, creating a metadata component for the remaining unused memory.

2.3 Miscellaneous

The myallocate function first gets a pointer to the 1st page of the current thread by calling getHead. It traverses to a free block of sufficient size to allocate the memory request. If there are no free blocks of sufficient size, it checks to see if there are enough remaining pages to satisfy the request. If so, it gets free pages first from main memory, and then from the swap file, until there is enough memory to satisfy the request, swapping the free pages into the next adjacent spot to pages owned by the current thread.

2.4 Testing

The memory manager and scheduler were compiled and tested on the iLab machines. The program was able to successfully serve a sequence of malloc calls followed by free calls, as well as a sequence of malloc calls interleaved with free calls. It was also able to successfully perform repeated malloc and free calls from different threads. It is also capable of allocating large memory requests that require multiple free pages at once.