Project 0

UNIX Systems Programming and User-Level Memory Allocation

Project 0 Part A is due Tuesday September 2rd at 11:59pm.

Project 0 Part B is due Tuesday September 9th at 11:59pm.

Start early. You will not finish if you start the night before the project is due.

You will submit your solution to your git repository.

Put Part A into prj0/parta and Part B into prj0/partb.

There are two attachments with given code (see website):

Part A: parta.tar.bz2

Part B: pintos.tar.bz2

Introduction

When we talk about executing code on a computer we make a distinction between application code (user-level code), and OS code (kernel-code). With the exception of this project, most of our work will be to extend the kernel code of the Pintos operating system. However, for this part of project 0 will look at some systems programming issues at the user-level. In Part A you will get experience with some of the standard UNIX system calls to construct parallel programs and in Part B, you will write a user-level memory allocator.

Part A: UNIX Systems Programming

The interface between user-level processes and the OS kernel is called the system call interface. These system calls are used by user processes to access OS supplied functionality such as process management, file creation/reading/writing, date/time access, interprocess communication, and several other I/O related operations. In this part of the project you are going to get some experience with using some of the standard UNIX system calls.

For this part of the project you will develop two simple parallel programs using UNIX processes, files, and pipes. Your programs will find the minimum and maximum value of a randomly generated list of integers. You are provided with a sequential version of the program and you need to construct two parallel versions as described below.

Sequential Min and Max

The given sequential version for finding min and max values in an array is called `findminmax_seq.c`. This program contains two functions: `find_min_and_max()` and `main()`.

The `find_min_and_max()` function is passed a pointer to an array of integers, `subarray`, and the size of the array, `n`. The array is scanned and working `min` and `max` values are updated when appropriate. When the function is finished, the final `min` and `max` values are passed back to the caller using a `results struct`.

The main() function accepts two command-line arguments: a seed value and the size of the random array to generate. The seed value is used by the random number generator. If you want to test with the same randomly generated array, you should use the same seed value from run to run. To test different arrays, use different seed values. This will be useful when testing your parallel versions for correctness. For example you can invoke findminmax_seq as follows:

```
$ findminmax_seq 1 10000
```

This will use a seed value of 1 and generate an array of size 10000.

The memory for the random array is dynamically allocated with malloc(). Once the new array is populated with random values, the find_min_and_max() function is called to find the min and max values. You will notice that some macros are invoked in order to record the execution time of the function call. These will useful to measure the performance improvements from the parallel versions.

Parallel Versions

Your parallel versions of findminmax will use UNIX processes to parallelize the work in finding the min and max values in the randomly generated array of integers. To do this, your new findminmax program will take an additional command line parameter: nprocs. So, the new usage syntax will be:

```
findminmax_par <seed> <arraysize> <nprocs>
```

Your parallel versions will use fork() system call to create nprocs processes, where each process will process arraysize / nprocs integers in the array. Your parallel versions should still populate the array with random numbers in a sequential fashion, just as in findminmax_seq.c. You should just have to make changes to main(), you should be able to leave find_min_and_max() unchanged.

You will have each UNIX process find the min and max values of a subarray of the main array. However, you will need to return these partial results back to the original process in order to find the final min and max values. Your two parallel versions will take different approaches to passing the partial results back to the main process. In the first version you will put partial results into files and have the main process read all the result files to compute the final min and max values. In the second version you will use UNIX pipes to pass the partial results back to the main process. See below for more details.

If you code this right, you should be able to reuse find_min_and_max() in the main process in order to process the partial results and find the final min and max values.

Parallel Min and Max Verison 1 (results communicated via files)

One useful approach to allow separate processes to communicate data is to do so using the file system. In your first parallel version of findminmax, you will have the subprocesses write partial results to intermediate files. Once the subprocesses are finished, the main processes will read in the partial results from the files to compute the final results. Your main process should delete the intermediate partial result files before exiting so that you do not leave the files in the file system. However, for debugging purposes you may want to leave the files initially.

You will have to pick a naming convention and data file format for the intermediate files. You can choose an ASCII or binary format for the intermediate files.

Parallel Min and Max Version 2 (results communicated via pipes)

In UNIX, processes running on the same machine can communicate using pipes. In your second parallel version you should have the subprocesses communicate partial results back to the main process using pipes. You should create one pipe for each subprocess. Similar to the first parallel version, you will need to pick a data format, as pipes are designed to allow processes to communicate arbitrary byte sequences.

Measurements

You should conduct the following experiments on a lab machine in Harney 530:

findminmax_seq 1 400000000
findminmax_par1 1 400000000 1
findminmax_par1 1 400000000 2
findminmax_par1 1 400000000 4
findminmax_par2 1 400000000 1
findminmax_par2 1 400000000 2
findminmax_par2 1 400000000 4

Put your results into a README file in your final submission in the parta directory.

Part B: A User-level Memory Allocator

Introduction

Pintos, despite being a state-of-the-art OS in most respects, currently lacks a user-level memory allocator. Programs running on Pintos cannot make use of `malloc()` and `free()` (the C equivalents of C++'s new and delete), which obviously is a sad state of affairs.

In this part of the project, you will implement a user-level memory allocator for Pintos. Along the way, you will learn about user-level memory allocation strategies and you'll learn how to use some of Pintos's code. A memory allocator divvys up a large, continuous piece of memory into smaller pieces, called blocks. Blocks can be of different sizes. To allocate memory, an allocator needs to find a free block of sufficient size. If memory is deallocated (freed), the block in which the memory was contained is returned to the pool of free memory. You will develop your solutions within the Pintos source tree. You can download a copy of the Pintos source tree here: pintos.tar.bz2.

Using Lists

You should use a linked list to keep track of free blocks. The free list should be sorted by address. Initially, all available memory is kept in a single, large block that is added to the free list. You should use Pintos's list implementation in `lib/kernel/list.c|h` for this project. You should read these two files to familiarize yourself with the API. In particular, pay attention to the functions it provides to maintain sorted lists. You will use this implementation in all future Pintos projects.

Interface

Here is the interface to your memory allocator (this is from the given file memalloc.h):

```
/* Initialize memory allocator to use 'length' bytes of memory at 'base'. */
void mem_init(uint8_t *base, size_t length);

/* Allocate 'length' bytes of memory. */
void * mem_alloc(size_t length);

/* Free memory pointed to by 'ptr'. */
void mem_free(void *ptr);

/* Return the number of elements in the free list. */
size_t mem_sizeof_free_list(void);

/* Dump the free list. */
void mem_dump_free_list(void);
```

The mem_init() function initializes your memory allocator. It requires a pointer to a continuous chunk of memory (more later) and the number of bytes in the continuous chunk.
The mem_alloc() and mem_free() functions work just like malloc() and free() except that all allocations and deallocations use the global memory specified in the call to mem_init(). Your implementation will need to keep track of all allocations and deallocations. You will maintain a free list for bookkeeping and it should be maintained in increasing memory address order (see below). Initially, the free list will contain one memory block that is the size of the global memory. Each time mem_alloc() or mem_free() are called, the free list should be updated appropriately. The mem_alloc() function should use a first-fit approach to finding a free block to accommodate a request (see below). If an allocation request cannot be satisfied, mem_alloc() should return NULL.

The mem_sizeof_free_list() function should return the number of elements on the free list.
The mem_dump_free_list() function should output a human readable list of free blocks. For each block the output should include the start address of the block and the length, in addition each block should appear on a separate line. The format should be:

```
address length
```

For example:

```
0x500120 1048576
0x600120 1048576
```

Allocation and First-Fit

Your allocator should use a first-fit strategy. If a request is made to allocate a certain number of bytes, your allocator should look for the first available block that has a size large enough to accommodate the request. If the block is larger than the memory that is requested, it must be split. To split a block, you should shorten the block's length by the amount you'll need for the to-be-allocated memory. The to-be-allocated memory comes from the top of the free block. In this way, you do not need to manipulate the free list when splitting - the block to-be-split stays in the free list. To remember the length of the block you allocated, you should prepend a used block header to the block you are allocating. This header contains the length of the block you are allocating. Make sure you account for its size when computing how many bytes to cut out of a given block. A block is not split if the portion that would remain is too small to be kept on the free list. Allocated blocks must be at least as large as a free block header - otherwise, if a block is freed, it would be impossible to reinsert that block into the

free list. If an allocation request is small, you will have to round it up accordingly.

Be sure to keep the free list sorted in order of increasing addresses.

Your allocator should return a pointer to the beginning of the usable memory inside the allocated block. Your allocator should return NULL if it cannot satisfy an allocation request. Even though first-fit may seem like a strategy too simple to be useful, it is actually a pretty good one. Other strategies include best-fit and next-fit. A next-fit allocator picks the next available block like first-fit, but does not start over when a new request arrives. Next-fit generally does worse than first-fit. A best-fit strategy looks for the block that fits best, leaving the smallest remainder or none. This can have the unfortunate consequence that a lot of small memory blocks are returned to the free list. Eventually, there are many small blocks on the free list, which creates fragmentation. Fragmentation makes it impossible to satisfy a request for memory, even though the total amount of free memory is still larger than the requested amount. Different allocation strategies cause different amounts of fragmentation. Unfortunately, there is no universal best solution - for any strategy, you can construct a sequence of allocations and deallocations that creates fragmentation. In practice, for many types of workloads, best-fit and first-fit perform roughly the same, so the added overhead of best-fit does not always pay off.

Deallocation and Coalescing

If memory is freed, you must find the beginning of the block of memory that contains the address of the pointer passed to the free routine. That block of memory must be added to the free list. In addition, you'll have to coalesce the free list: if the blocks to the left and/or right of the block being freed are also free, they must be merged into a single block. You should implement a function that reports the length of the free list.

Assumptions

You may assume that the allocation function is called with a size that is a multiple of 4. You may assume that `mem_init()` is called with a size that is a multiple of 4 and that is greater or equal to 16. Our test harness will invoke your memory allocator and execute a series of allocation and deallocation requests. Finally, it will deallocate all allocated memory and check that you properly coalesced the free memory into a single, large block.

Thread Safety

Your memory allocator should be thread-safe. That is, it should be able to handle being invoked by multiple threads concurrently. To accomplish this, you must protect its data structures so that only one thread can access them at any given point in time. In the Pintos kernel, you would use `lock_acquire()`/`lock_release()` for this purpose. For now, we will use the POSIX thread API which is provided by Linux. You need to use the functions `pthread_mutex_lock()` and `pthread_mutex_unlock()`. You will also need to initialize the mutex or mutexes you will be using. To see how to do that, read the man pages for `pthread_mutex_lock(P)` and `pthread_mutex_init(P)`. These are the only pthread functions you will need to use. We recommend that you first develop and test your allocator with one thread, and then add the necessary protection to pass the multi-threaded part of the test harness. (Tests 1, 2, and 4 are run by a single thread, only Test 3 exercises the allocator concurrently by multiple threads.)

Instructions

For Part B, you will work in the pintos/src/prep directory for this assignment. Copy the pintos source tree into a directory of your choice:

```
$ mkdir cs326
$ cd cs326
$ mkdir prj0
$ cd prj0
$ mkdir partb
$ cd partb
$ tar xvjf <path>/pintos.tar.bz2
```

You must add a file called `memalloc.c` to that directory that implements the memory allocator's interface, described in `memalloc.h`. Please read this file for specifics. `memalloc.h` also contains struct free_block and struct used_block definitions you may use to represent free and used blocks of memory, respectively. You build the test harness using make. This will build the test_mem executable. `./test_mem` will run it.

Grading

This assignment will count for 100 points. Unlike for future projects, you are not required to submit a design document with this project. The points break down as follows:

Part A

20 points parallel version 1
20 points parallel version 2

Part B

10 points: passing Test 1.
10 points: passing Test 1 and 2.
10 points: passing Test 1, 2, and 3 (includes proper use of locks for thread safety)
10 points: passing Test 1, 2, 3, and 4.
10 points: proper use of Pintos's list implementation.

Both Parts

10 points: adherence to coding standards outlined in Pintos project documentation

FAQ

Part B

Q: A production version of first-fit wouldn't use a simple doubly-linked list, wouldn't it?
A: No, it would probably use a more scalable data structure. But a linear list is acceptable for this pro ject.

Q: What extra credit is there?
A: For extra credit, implement best fit and compare the fragmentation produced by first-fit to the fragmentation produced by best-fit for the test harness's workload. Develop a reasonable measure by which to capture fragmentation quantitatively. Produce a report that outlines your results in a concise manner.

Q: Can we make changes to test_mem.c?
A: No. We will use the test_mem.c that we provided. Your implementation should be in memalloc.c which you'll write. You may make changes to memalloc.h if you deem them necessary.

Q: Given the void* pointer passed to mem_free, how do I get a pointer to the surrounding memory block?
A: Consider using the offsetof macro. Other options exist as well, but make sure your code compiles without warnings.