Computer Operating System Experiment

Laboratory 7

Memory Management

Objective:

- Learn to design and to implement a custom memory allocator:

  The operating system provides mechanisms for dynamic allocation in contiguous memory spaces. There are multiple criteria to assess how well these mechanisms work, including speed of execution and the minimization of external fragmentation (to that end, you learned of different policies for allocation decisions, such as first- fit, best-fit, and worst-fit). In this lab, you will design and implement a memory allocation system to work in user space and apply the concepts you have studied so far.

- In this lab, you will write your own dynamic memory allocator called allocator() that you should be able to use in place of the standard malloc() utility.

Equipment:

VirtualBox with Ubuntu Linux

Methodology:

Program and answer all the questions in this lab sheet.

1. Memory management

Memory management is a form of resource management applied to computer memory. The essential requirement of memory management is to provide ways to dynamically allocate portions of memory to programs at their request, and free it for reuse when no longer needed. This is critical to any advanced computer system where more than a single process might be underway at any time.

We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory. In contiguous memory allocation, each process is contained in a single section of memory that is contiguous to the section containing the next process. In the variable-partition scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes and is considered one large block of available memory, a hole. Eventually, as you will see, memory contains a set of holes of various sizes.

This procedure is a particular instance of the general dynamic storage allocation problem, which concerns how to satisfy a request of size n from a list of free holes. There are many solutions to this problem. The first-fit, best-fit, and

worst-fit strategies are the ones most commonly used to select a free hole from the set of available holes

- First fit. Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
- Best fit. Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
- Worst fit. Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

2. Experiments

2.1 Experiment 1: customed memory allocation

The implementation of a doubly-linked list abstract data type (ADT) in C is given to you. You may use this code with the usual guarantees of open software: The solution that is being given to you has undergone a good amount of testing, but there are no guarantees that it is absolutely perfect. You may want to read

the code to understand the implementation, you may want to test it further, or do anything else you find appropriate to develop the confidence you have in it.

Be sure to put the files you create for this lab in the appropriate directories in this source code tree! That is, new header files should go in include/, new source files should go in src/, etc. Note that you may want to modify the code of this doubly-linked list so as to have a little more information in the node structure than what is given to you in the original module. (See more about this point in the description of the deallocate function below.) If you modify the doubly-linked list code, you should want to modify the test suite in src/dlisttest.c and run these tests to ensure that the module runs correctly after your changes.

For this problem, you will create a custom memory allocator. Since your code will not be part of the operating system, it will execute in user mode. The general idea is that one who uses your allocator will include a header file which declares your functions and also link with the code that defines/implements them. A program using your allocator will have to get it initialized before any calls are made to use dynamically allocated memory; our standard prototype for the initialization is:

```
int allocator_init(size_t size);
```

This function will create and initialize two doubly-linked lists used for memory management: one which keeps track of the memory that is available (call it free_list) and another which keeps track of memory which is already in use

(call it allocated_list). All the memory manipulated by your allocator will reside in the heap of the process that uses it. When allocator_init starts, it will call the standard malloc to request a contiguous space of size bytes from the underlying operating system. The pointer received from malloc will be used to initialize a single node in your free_list; your allocated_list must start out empty. If the malloc call does not succeed, this function must return -1, otherwise, it must return 0.

Ultimately, a custom memory allocator will need an API similar to what you have for malloc and free, functions that provide dynamic memory allocation in C. With that in mind, we define the API for a function to allocate memory and another to deallocate.

```
void *allocate (size_t size);
```

Equivalent to malloc. This function will traverse the free_list and find a contiguous chunk of memory that can be used to satisfy the request of an area of size bytes. If the caller makes a request for memory that is larger than what your allocator can honor, this function must return NULL. Your allocate function must be flexible enough to allow for different allocation policies, namely first-fit, best-fit, and worst-fit. You should probably create three functions, one for each of these policies, which are used internally by allocate and are not visible to the user of your custom memory allocator. Having those functions allows you to make easy modifications to the policy used by allocate. You will want to design your code so that it is easy run experiments with different allocation policies, so

think carefully about how you will define your functions prototypes. The design of these functions' API and their implementation (obviously) is up to you.

```
int deallocate(void *ptr);
```

Equivalent to free. This function will use ptr to find the corresponding node in the allocated_list and free up that chunk of memory. If the caller attempts to deallocate a pointer that cannot be found in the allocated_list, this function must return -1, otherwise it must return 0. To make your development process easier, at first, you can simply move the deallocated memory from the allocated_list to the free_list. Note that just as the C library's free, deallocate does not ask you for the size of the memory you are returning to the system. Think about how you can make your custom allocator keep track of the size of each allocated chunk of memory — the cleanest solution might require you to change the code of the doubly-linked list given to you.

The entire code of your custom memory allocator will be encapsulated in two files:

```
allocator.h
allocator.c
```

The allocator.h file is shown below.

```
#include <stdint.h>
#include <stdbool.h>
#include "dnode.h"
#include "dlist.h"
/*
initialize allocated and free list
*/
int allocator_init(size_t size, int typee);
/*
malloc()
*/
void* allocate(size_t size);
/*
free()
*/
int deallocate(void *ptr);

/*
Use First-Fit policy
*/
void* allocateFirstFit(size_t size);
/*
Use Best-Fit policy
*/
void* allocateBestFit(size_t size);
/*
Use Worst-Fit policy
*/
void* allocateWorstFit(size_t size);
/*
Show state of free-list
*/
void showFreeList();
/*
Show state of allocated-list
*/
void showAllocatedList();
/*
Modify free-list to have one node
*/
void modifyFreeList();
```

Requirements：

1) You must include the **allocator.h** and implement **allocator.c** using first fit, best fit and worst fit algorithms respectively.

2) To test your implementation, you will need to create one or more test programs, each with their own main() function, to exercise your memory allocator. If you have a single file with all your tests, submit it as memory-test.c.

2.2 Experiment 2: observation the fragmentation

Create a new function in your allocator to compute the average fragmentation created in memory after repeated, randomly interleaved calls to your allocate and deallocate functions. Before you get down to coding this function, though, think about how you will implement it and write here your algorithm in the form of an algorithm in pseudo-C code. The signature for this function should be something like:

```
double average_frag();
```

You can get the average fragmentation using below:

average_frag = totalFreeSpace / the # holes;

Finally, once you have the algorithm for average_frag, implement the corresponding function in the C module that contains your allocator. That is, you will be changing your **allocator.h** and **allocator.c** files to include the new function.

You need to implement frag-eval.c. Modify your new file so that its main function accepts command line parameters as in the usage guide below:

frag-eval [algorithm] [requests]

where:

- algorithm (integer) can take only the tree values which select different allocation policies:0 (means first-fit), 1 (means best-fit), 2 (means worst-fit)
- requests (integer) specifies the number of dynamic memory allocation

requests to simulate before the evaluation of fragmentation is performed.

Your main function must implement some pattern of allocate and deallocate requests to exercise the functionality of your allocator. Consider the algorithm given below in pseudo-C code:

```
//initialize your custom memory allocator to work with a pool of 1024*1024 bytes (1M)
srandom(seed); // initialize PRNG, or Pseudo-Random Number Generator
int r=0;
void *p = NULL;
while (r < requests) {
   s = a random number between 100 and 10000;
   p = allocate(algorithm, s)
   // this will require you to change your allocate function
   // so that it accepts the algorithm parameter to select an
   // allocation policy
   deallocate(p);
}
```

Requirements：

1) Choose a value of R of runs to perform and also choose R different seeds to provide for the generation of pseudo-random numbers. (Consider only values of R >= 30)

2) For each of the three allocation policies (first-fit, best-fit, and worst-fit), complete R runs using the selected seeds and compare the total

fragmentation and the average fragmentation.