西北工業大學

NORTHWESTERN POLYTECHNICAL UNIVERSITY

Lab 7 Report

Report Subject: OS Experiment - Lab 7

Student ID: 2018380039

Student Name: Dikshya Kafle

Submission Date: 2021/11/25

Computer Operating System Experiment

Computer Operating System Experiment

Laboratory 7

Memory Management

1) Objective:

- Learn to design and to implement a custom memory allocator
- Write a dynamic memory allocator called allocator() that you should be able to use in place of the standard malloc() utility.

2) Equipment:

- VirtualBox with Ubuntu Linux

3) Experiments:

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.

Experiment 1: Customed memory allocation

We include the allocator.h file and use the first fit, best fit, and worst fit algorithms to create the allocator.c file. To put our implementation to the test, we write three programs, each with its own main() function, to put the memory allocator to the test:

- **memory-test1.c:** The First-Fit approach is put to the test in this program. The First-fit allocates the first large enough hold it finds for the task. First-fit is the quickest allocation strategy and performs better with storage than worst-fit, although it suffers from external fragmentation.

```
dikshya@dikshya-VirtualBox:~/Desktop/Lab 7 OS$ ./memory-test1
Calling allocator_init to initialize allocator and linked lists
Free List: 1500
Allocated List:

Allocate a block of size 100 using the first-fit method: success!
Free List: 1400
Allocated List: 100

Allocate a block of size 200 using the first-fit method: success!
Free List: 1200
Allocated List: 100 200

Allocate a block of size 300 using the first-fit method: success!
Free List: 900
Allocated List: 100 200 300

Allocate a block of size 400 using the first-fit method: success!
Free List: 500
Allocated List: 100 200 300 400

Allocate a block of size 500 using the first-fit method: success!
Free List: 0
Allocated List: 100 200 300 400 500

Deallocate node of size 100: Success!
Free List: 0 100
Allocated List: 200 300 400 500
dikshya@dikshya-VirtualBox:~/Desktop/Lab 7 OS$
```

❖ **memory-test2.c:** The Best-Fit method is put to the test in this program. The Best-fit algorithm selects the smallest hole that is large enough to accommodate the process. It has the advantage of being faster and having better storage allocation than worst-fit, but it suffers from external fragmentation and must search the complete list. As a result, it is slower than the first-fit method.

```
dikshya@dikshya-VirtualBox:~/Desktop/Lab 7 OS$ ./memory-test2
Calling allocator_init to initialize allocator and linked lists
Free List: 1500
Allocated List:

Allocate a block of size 100 using the best-fit method: Success!
Free List: 1400
Allocated List: 100

Allocate a block of size 200 using the best-fit method: Success!
Free List: 1200
Allocated List: 100 200

Allocate a block of size 300 using the best-fit method: Success!
Free List: 900
Allocated List: 100 200 300

Allocate a block of size 400 using the best-fit method: Success!
Free List: 500
Allocated List: 100 200 300 400

Allocate a block of size 500 using the best-fit method: Success!
Free List: 0
Allocated List: 100 200 300 400 500

Deallocate node of size 100: Success!
Free List: 0 100
Allocated List: 200 300 400 500
dikshya@dikshya-VirtualBox:~/Desktop/Lab 7 OS$
```

❖ **memory-test3.c:** The Worst-Fit method is put to the test in this program. The Worst-fit method assigns the process the largest available hole. This policy has the benefit of reducing the amount of minor gaps between memory areas. However, because it must search through the full list, the worst-fit is sluggish. It is the slowest and uses the least amount of storage of the three policies.

```
dikshya@dikshya-VirtualBox:~/Desktop/Lab 7 OS$ ./memory-test3
Calling allocator_init to initialize allocator and linked lists
Free List: 1500
Allocated List:

Allocate a block of size 100 using the worst-fit method: Error in allocate, wrong policy number!
success!
Free List: 1500
Allocated List: 100

Allocate a block of size 200 using the worst-fit method: Error in allocate, wrong policy number!
success!
Free List: 1500
Allocated List: 100 200

Allocate a block of size 300 using the worst-fit method: Error in allocate, wrong policy number!
success!
Free List: 1500
Allocated List: 100 200 300

Allocate a block of size 400 using the worst-fit method: Error in allocate, wrong policy number!
success!
Free List: 1500
Allocated List: 100 200 300 400

Allocate a block of size 500 using the worst-fit method: Error in allocate, wrong policy number!
success!
Free List: 1500
Allocated List: 100 200 300 400 500

Deallocate node of size 100: Success!
Free List: 1500 100
Allocated List: 200 300 400 500
dikshya@dikshya-VirtualBox:~/Desktop/Lab 7 OS$
```

Experiment 2: Observation the fragmentation

External fragmentation is a sort of fragmentation that our memory allocator can handle. This could happen if we receive a request for memory space and there is adequate space available, but the memory spaces are not contiguous.

We add a new function to the allocator to compute the average memory fragmentation caused by repeated, randomly interleaved allocation and deallocation calls. The following pseudo-code can be used to describe the function's algorithm:

frag-eval(algo,seed,requests) {

*allocator_init(1024*1024)*

srandom(seed);

int r = 0;

*void *p = NULL;*

while(r<requests) {

s = a random int between 100 and 10000

p = allocate(algo,s)

deallocate(p)

r++

}

return average_frag()

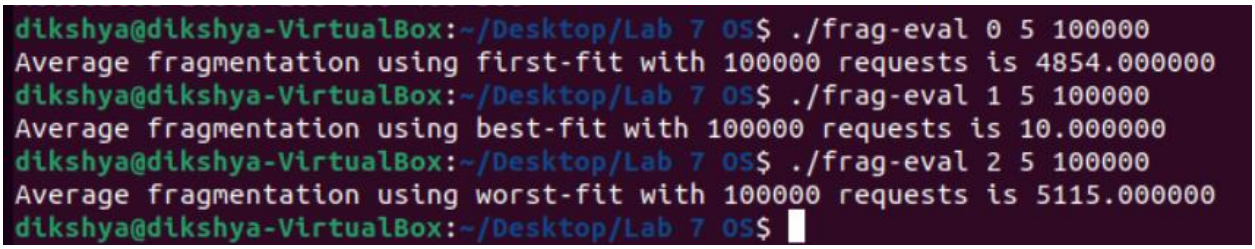
}

As shown in the example below, the main function accepts command line parameters:

./frag-eval [Algorithm] [Seed] [Requests]

- Only the tree values that pick distinct allocation strategies can be used in the algorithm (integer): 0 (means first-fit), 1 (means best-fit), 2 (means second-best-fit) (means worst-fit).
- Requests (integer) indicates how many dynamic memory allocation requests should be simulated before fragmentation is evaluated.
- Seed is used to start the random number generator.

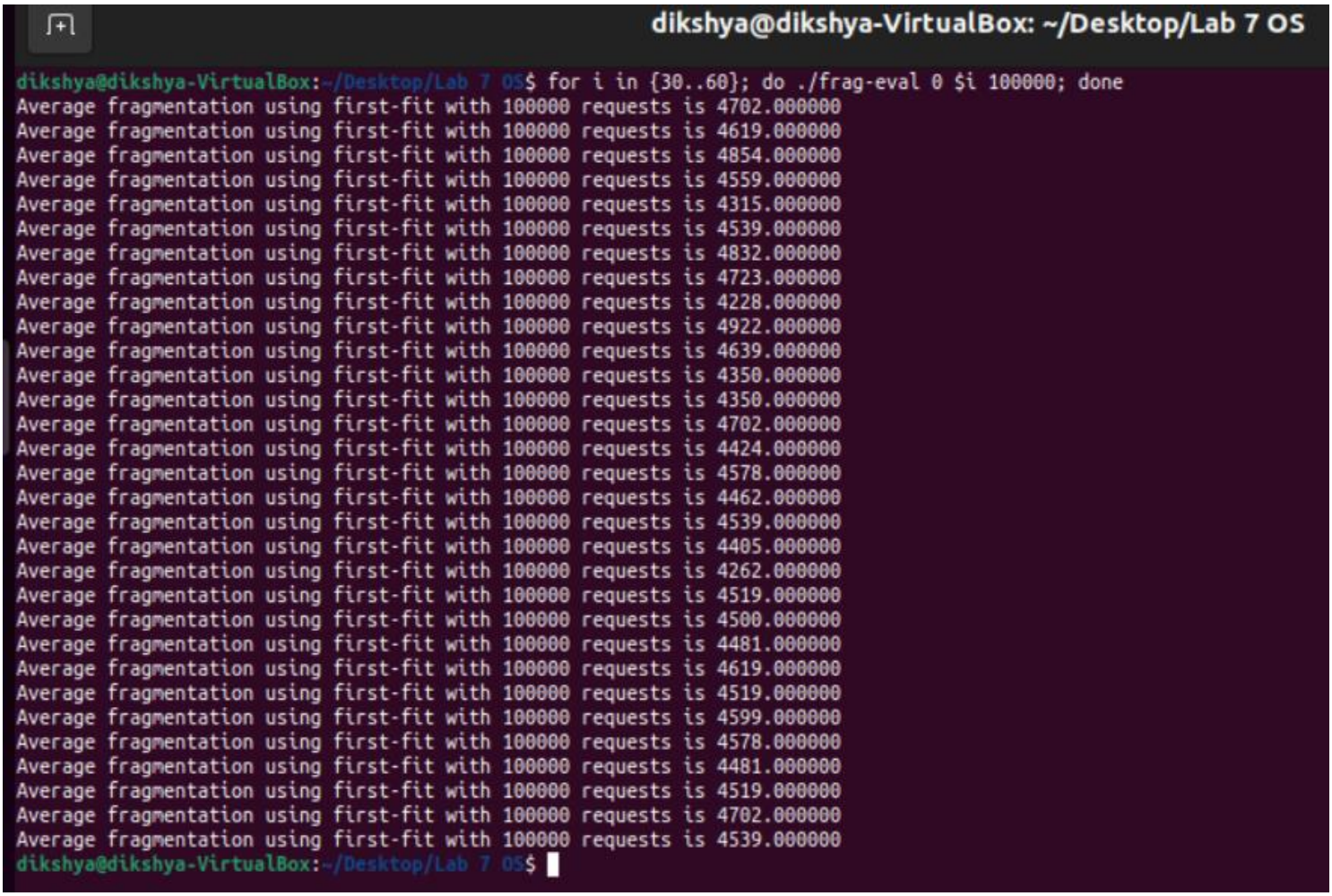
For each of the three memory allocation options, here's an example of how to run our main function:



After that, we run our program several times. We choose an R-value for the number of runs to run, as well as R distinct seeds for the production of pseudo-random numbers. (We only examine R values greater than 30; R >= 30). Complete R runs with the selected seeds for each of the three allocation rules (first-fit, best-fit, and worst-fit). To accomplish so, we utilize the following script:

"for i in {30..60}; do ./frag-eval <policy_number> $i <requests>; done"

❖ **First-Fit:**


dikshya@dikshya-VirtualBox: ~/Desktop/Lab 7 OS
dikshya@dikshya-VirtualBox:~/Desktop/Lab 7 OS$ for i in {30..60}; do ./frag-eval 0 $i 100000; done
Average fragmentation using first-fit with 100000 requests is 4702.000000
Average fragmentation using first-fit with 100000 requests is 4619.000000
Average fragmentation using first-fit with 100000 requests is 4854.000000
Average fragmentation using first-fit with 100000 requests is 4559.000000
Average fragmentation using first-fit with 100000 requests is 4315.000000
Average fragmentation using first-fit with 100000 requests is 4539.000000
Average fragmentation using first-fit with 100000 requests is 4832.000000
Average fragmentation using first-fit with 100000 requests is 4723.000000
Average fragmentation using first-fit with 100000 requests is 4228.000000
Average fragmentation using first-fit with 100000 requests is 4922.000000
Average fragmentation using first-fit with 100000 requests is 4639.000000
Average fragmentation using first-fit with 100000 requests is 4350.000000
Average fragmentation using first-fit with 100000 requests is 4350.000000
Average fragmentation using first-fit with 100000 requests is 4702.000000
Average fragmentation using first-fit with 100000 requests is 4424.000000
Average fragmentation using first-fit with 100000 requests is 4578.000000
Average fragmentation using first-fit with 100000 requests is 4462.000000
Average fragmentation using first-fit with 100000 requests is 4539.000000
Average fragmentation using first-fit with 100000 requests is 4405.000000
Average fragmentation using first-fit with 100000 requests is 4262.000000
Average fragmentation using first-fit with 100000 requests is 4519.000000
Average fragmentation using first-fit with 100000 requests is 4500.000000
Average fragmentation using first-fit with 100000 requests is 4481.000000
Average fragmentation using first-fit with 100000 requests is 4619.000000
Average fragmentation using first-fit with 100000 requests is 4519.000000
Average fragmentation using first-fit with 100000 requests is 4599.000000
Average fragmentation using first-fit with 100000 requests is 4578.000000
Average fragmentation using first-fit with 100000 requests is 4481.000000
Average fragmentation using first-fit with 100000 requests is 4519.000000
Average fragmentation using first-fit with 100000 requests is 4702.000000
Average fragmentation using first-fit with 100000 requests is 4539.000000
dikshya@dikshya-VirtualBox:~/Desktop/Lab 7 OS$

- **Best-Fit:**

```
dikshya@dikshya-VirtualBox:~/Desktop/Lab 7 OS$ for i in {30..60}; do ./frag-eval 1 $i 100000; done
Average fragmentation using best-fit with 100000 requests is 10.000000
Average fragmentation using best-fit with 100000 requests is 10.000000
Average fragmentation using best-fit with 100000 requests is 10.000000
Average fragmentation using best-fit with 100000 requests is 10.000000
Average fragmentation using best-fit with 100000 requests is 10.000000
Average fragmentation using best-fit with 100000 requests is 10.000000
Average fragmentation using best-fit with 100000 requests is 10.000000
Average fragmentation using best-fit with 100000 requests is 10.000000
Average fragmentation using best-fit with 100000 requests is 10.000000
Average fragmentation using best-fit with 100000 requests is 10.000000
Average fragmentation using best-fit with 100000 requests is 10.000000
Average fragmentation using best-fit with 100000 requests is 10.000000
Average fragmentation using best-fit with 100000 requests is 10.000000
Average fragmentation using best-fit with 100000 requests is 10.000000
Average fragmentation using best-fit with 100000 requests is 10.000000
Average fragmentation using best-fit with 100000 requests is 10.000000
Average fragmentation using best-fit with 100000 requests is 10.000000
Average fragmentation using best-fit with 100000 requests is 10.000000
Average fragmentation using best-fit with 100000 requests is 10.000000
Average fragmentation using best-fit with 100000 requests is 10.000000
Average fragmentation using best-fit with 100000 requests is 10.000000
Average fragmentation using best-fit with 100000 requests is 10.000000
Average fragmentation using best-fit with 100000 requests is 10.000000
Average fragmentation using best-fit with 100000 requests is 10.000000
Average fragmentation using best-fit with 100000 requests is 10.000000
Average fragmentation using best-fit with 100000 requests is 10.000000
Average fragmentation using best-fit with 100000 requests is 10.000000
Average fragmentation using best-fit with 100000 requests is 10.000000
Average fragmentation using best-fit with 100000 requests is 10.000000
Average fragmentation using best-fit with 100000 requests is 10.000000
Average fragmentation using best-fit with 100000 requests is 10.000000
dikshya@dikshya-VirtualBox:~/Desktop/Lab 7 OS$
```

- **Worst-Fit:**

```
dikshya@dikshya-VirtualBox:~/Desktop/Lab 7 OS$ for i in {30..60}; do ./frag-eval 2 $i 100000; done
Average fragmentation using worst-fit with 100000 requests is 4877.000000
Average fragmentation using worst-fit with 100000 requests is 4766.000000
Average fragmentation using worst-fit with 100000 requests is 5017.000000
Average fragmentation using worst-fit with 100000 requests is 4702.000000
Average fragmentation using worst-fit with 100000 requests is 4481.000000
Average fragmentation using worst-fit with 100000 requests is 4681.000000
Average fragmentation using worst-fit with 100000 requests is 5090.000000
Average fragmentation using worst-fit with 100000 requests is 4899.000000
Average fragmentation using worst-fit with 100000 requests is 4387.000000
Average fragmentation using worst-fit with 100000 requests is 5090.000000
Average fragmentation using worst-fit with 100000 requests is 4723.000000
Average fragmentation using worst-fit with 100000 requests is 4481.000000
Average fragmentation using worst-fit with 100000 requests is 4500.000000
Average fragmentation using worst-fit with 100000 requests is 4877.000000
Average fragmentation using worst-fit with 100000 requests is 4559.000000
Average fragmentation using worst-fit with 100000 requests is 4766.000000
Average fragmentation using worst-fit with 100000 requests is 4578.000000
Average fragmentation using worst-fit with 100000 requests is 4723.000000
Average fragmentation using worst-fit with 100000 requests is 4539.000000
Average fragmentation using worst-fit with 100000 requests is 4443.000000
Average fragmentation using worst-fit with 100000 requests is 4681.000000
Average fragmentation using worst-fit with 100000 requests is 4723.000000
Average fragmentation using worst-fit with 100000 requests is 4660.000000
Average fragmentation using worst-fit with 100000 requests is 4899.000000
Average fragmentation using worst-fit with 100000 requests is 4766.000000
Average fragmentation using worst-fit with 100000 requests is 4766.000000
Average fragmentation using worst-fit with 100000 requests is 4681.000000
Average fragmentation using worst-fit with 100000 requests is 4619.000000
Average fragmentation using worst-fit with 100000 requests is 4681.000000
Average fragmentation using worst-fit with 100000 requests is 4969.000000
Average fragmentation using worst-fit with 100000 requests is 4899.000000
dikshya@dikshya-VirtualBox:~/Desktop/Lab 7 OS$
```

Conclusion:

The average fragmentation is clearly lower when using best-fit rather than first-fit or worst-fit. The average fragmentation of first-fit is lower than that of worst-fit, but not as low as that of best-fit. As a result, the best-fit allocation strategy performed the best in our simulation.

In general, the worst-fit is thought to be the least memory-efficient, while the best-fit is thought to be the most memory-efficient. Although first-fit is quicker than worst-fit and best-fit, it may not provide the optimum memory solution. When it comes to memory optimization, first-fit and best-fit are usually similar; neither is known to be better than the other. However, we can observe that best-fit has better memory optimization in this scenario.