HCMC University Of Technology
Faculty of Computer Science & Engineering

Lab 8 Contiguous Memory Allocation
Course: Operating Systems

November 25, 2020

Goal: This lab helps student to practice the operations of allocation on main memory in Operating System.

Content In detail, this lab requires student implement one of the memory allocation algorithms from the given source codes. We use these source code to emulate the allocation algorithm of memory using **First-Fit**. The other algorithm that student will practice is **Best-Fit**. Furthermore, student can implement **Worst-Fit** or **Next-Fit** algorithm at home.

Result After doing this lab, student can understand the idea of each allocation algorithm in memory and the concept of fragmentation.

Requirement Student need to review the theory of chapter named Main Memory.

1 Background

Initially, each process needs a separate memory space. Separate per-process memory space protects the processes from each other and is fundamental to having multiple processes loaded in memory for concurrent execution. To separate memory spaces, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses. We can provide this protection by using two registers, usually a base and a limit, as figure 1.1 shown. The

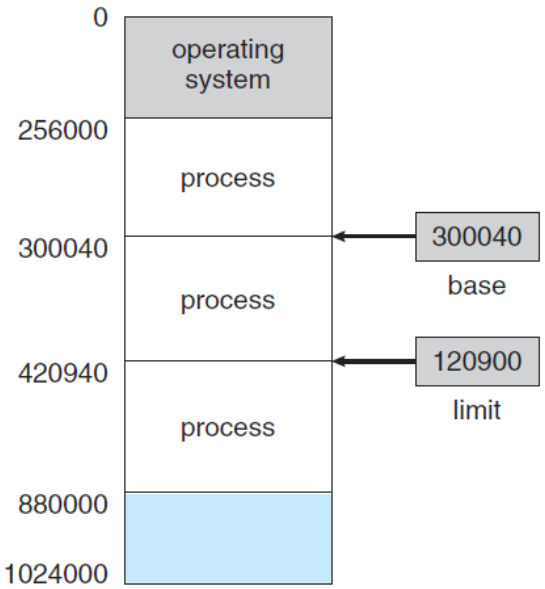

Figure 1.1: A base and a limit register define a logical address space.

base register holds the smallest legal physical memory address; the **limit register** specifies the size of the range.

1.1 Address binding

Usually, a program resides on a disk as a binary executable file. To be executed, the program must be brought into memory and placed within a process. Depending on the memory management in use, the process may be moved between disk and memory during its execution. The processes on the disk that are waiting to be brought into memory for execution form the **input queue**. Classically, the binding of instructions and data to memory addresses can be done at any step along the way:

1. Compile time
2. Load time
3. Execution time

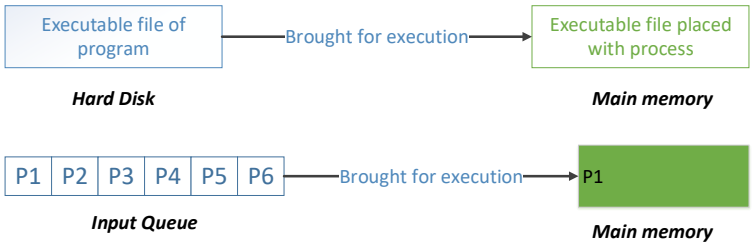

Figure 1.2: Process moves between disk and memory during execution.

1.2 Logical Versus Physical Address Space

An address generated by the CPU is commonly referred to as a **logical address**, whereas an address seen by the memory unit that is, the one loaded into the **memory-address register** of the memory is commonly referred to as a **physical address**. The compile time and load time address-binding methods generate identical logical and physical addresses.

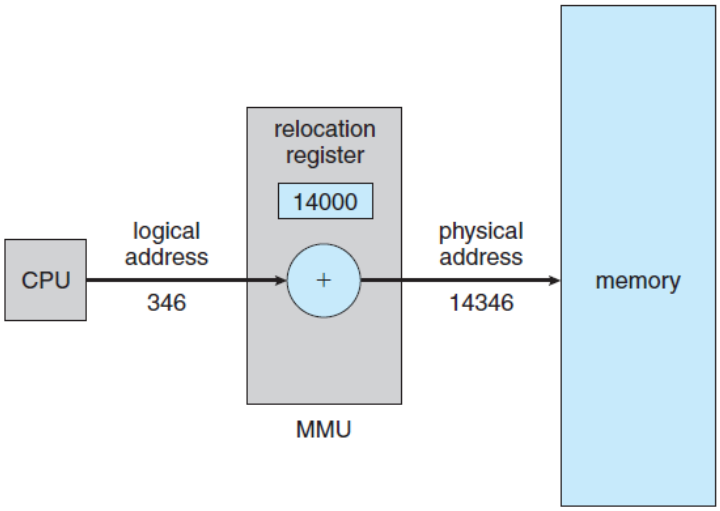

Figure 1.3: Dynamic relocation using a relocation register.

However, the execution time address binding scheme results in differing logical and physical addresses. The set of all logical addresses generated by a program is a **logical address space**. The set of all physical addresses corresponding to these logical addresses is a **physical address space**. Thus, in the execution-time address-binding scheme, the logical and physical address spaces differ. The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit (MMU)**.

1.3 Dynamic loading

The size of a process has thus been limited to the size of physical memory. To obtain better memory-space utilization, we can use dynamic loading. With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format. The main program is loaded into memory and is executed. The advantage of dynamic loading is that a routine is loaded only when it is needed.

1.4 Swapping

A process must be in memory to be executed. A process, however, can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution. Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus increasing the degree of multi-programming in a system.

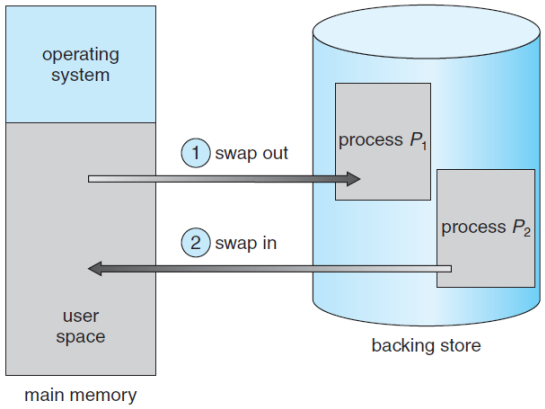

Figure 1.4: Swapping of two processes using a disk as a backing store.

1.5 Contiguous Memory Allocation

The main memory must accommodate both the operating system and the various user processes.We therefore need to allocate main memory in the most efficient way possible. The memory is usually divided into two partitions: one for the resident operating system and one for the user processes.

In general, as mentioned, the memory blocks available comprise a set of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. 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 Practice

The given source code emulates the first-fit algorithm of memory allocation in Operating System. You need to understand the content of each source file to implement the best-fit algorithm. In file named `main.c`, we simulate a process of allocating or de-allocating memory.

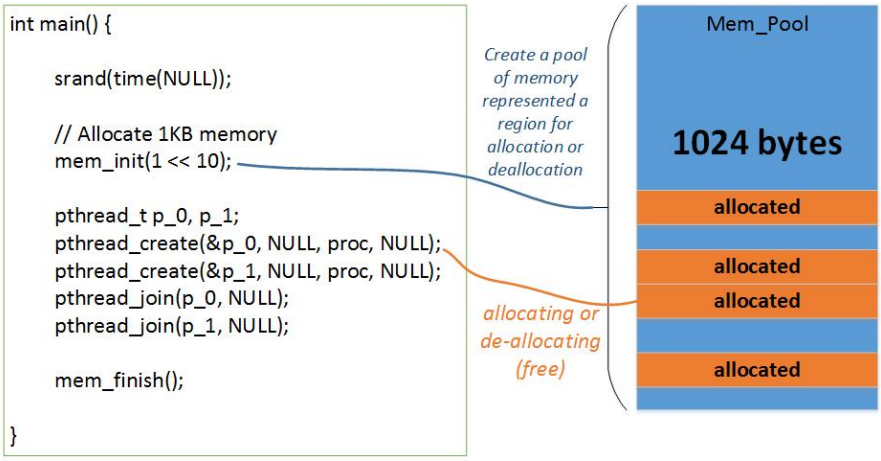

Figure 2.1: Diagram of simulating the process of allocation, deallocation in memory.

Firstly, the `main()` function need to create a region considered as a memory with the size of 1024 bytes. After that, this program creates 2 processes to simulate 2 running processes in OS. These two processes can require allocation or free randomly with sizes such as 16, 32, 64, 128 bytes. Therefore, student need to implement **allocation strategies** in memory which are described in `mem_alloc(size)` function. the given source implemented **First-Fit** allocator, student need to implement **Best-Fit** allocator.

```c
////////// main.c ////////////
#include "mem.h"
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <time.h>

#define ARRAY_SIZE 10

void * proc(void *args) {
        int i;
        int index = 0;
        char * mem[ARRAY_SIZE];
        for (i = 0; i < ARRAY_SIZE; i++) {
                if (rand() % 2) {
                        /* Allocate memory */
                        unsigned int size = 1 << ((rand() % 4) + 4);
                        mem[index] = mem_alloc(size);
                        if (mem[index] != NULL) {
                                index++;
                        }
                }else{
                        /* Free memory */
                        if (index == 0) {
                                continue;
                        }
                        unsigned char j = rand() % index;
                        mem_free(mem[j]);
                        mem[j] = mem[index - 1];
                        index--;
                }
        }
}

int main() {

        srand(time(NULL));

        // Allocate 1KB memory pool
        mem_init(1 << 10);

        pthread_t p_0, p_1;
        pthread_create(&p_0, NULL, proc, NULL);
```

```
44          pthread_create(&p_1, NULL, proc, NULL);
45          pthread_join(p_0, NULL);
46          pthread_join(p_1, NULL);
47
48          mem_finish();
49  }
```

The allocation algorithms have to be implemented in file `mem.c`. All comments and hints are described in this file.

```
1   #include "mem.h"
2   #include <stdlib.h>
3   #include <pthread.h>
4   #include <stdio.h>
5
6   void * mem_pool = NULL;
7
8   pthread_mutex_t lock;
9
10  struct mem_region {
11          size_t size;     // Size of memory region
12          char * pointer; // Pointer to the first byte
13          struct mem_region * next; // Pointer to the next region in
                the list
14          struct mem_region * prev; // Pointer to the previous region
                in the list
15  };
16
17  struct mem_region * free_regions = NULL;
18  struct mem_region * used_regions = NULL;
19
20
21  static void * best_fit_allocator(unsigned int size);
22  static void * first_fit_allocator(unsigned int size);
23
24  int mem_init(unsigned int size) {
25          /* Initial lock for multi-thread allocators */
26          return (mem_pool != 0);
27  }
28
29  void mem_finish() {
30          /* Clean preallocated region */
31          free(mem_pool);
32  }
33
```

```
34 void * mem_alloc(unsigned int size) {
35         pthread_mutex_lock(&lock);
36         // Follow is FIST FIT allocator used for demonstration only.
37         // You need to implment your own BEST FIT allocator.
38         // TODO: Comment the next line
39         void * pointer = first_fit_allocator(size);
40         // Commnent out the previous line and uncomment to next line
41         // to invoke best fit allocator
42         // TODO: uncomment the next line
43         //void * pointer = best_fit_allocator(size);
44
45         // FOR VERIFICATION ONLY. DO NOT REMOVE THESE LINES
46         if (pointer != NULL) {
47                 printf("Alloc [%4d bytes] %p-%p\n", size, pointer, (
                       char*)pointer + size - 1);
48         }else{
49                 printf("Alloc [%4d bytes] NULL\n");
50         }
51
52         pthread_mutex_unlock(&lock);
53         return pointer;
54 }
55
56 void mem_free(void * pointer) {
57         /* free memory */
58 }
59
60 void * best_fit_allocator(unsigned int size) {
61         // TODO: Implement your best fit allocator here
62         return NULL; // remember to remove this line
63 }
64
65 void * first_fit_allocator(unsigned int size) {
66         /* First fit example is shown in the given source code*/
67 }
```

One of important problems happened during memory allocation is **fragmentation**. Let's discuss about **fragmentation**. Student need to complete the **best-fit** allocator at the class.

3 Exercise

1. Given six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)? Rank the algorithms in terms of how efficiently they use memory.

2. Student write a short report that compares the advantages as well as disadvantages of the allocation algorithms, namely **First-Fit**, **Best-Fit**, **Worst-Fit**.