HCMC University Of Technology
Faculty of Computer Science & Engineering

– CO2018 Operating Systems – Fall 2018 –

Lab 8
Memory Allocation

October 26, 2018

Goal This lab helps student perceive the principle of memory allocation mechanisms as well as the concept of fragmentation.

Content After elaborating on fundamental knowledge, you will practice with some memory allocation algorithms including:

- First-fit
- Best-fit
- Worst-fit

Grading policy

- 30% in-class performance
- 70% BKeL submission

1 Literature review

Typically, 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. Each process acquires a particular memory space. Separate per-process memory space protects the processes from overlapping each other's area and is fundamental to having multiple processes loaded in a memory for concurrent execution. To achieve that, for a specific process, we need to determine a logical address space that it may access and to ensure that the process only accesses the assigned space. Figure 1.1 illustrates how to provide such logical address space to a process using a pair of **base** and **limit** register.

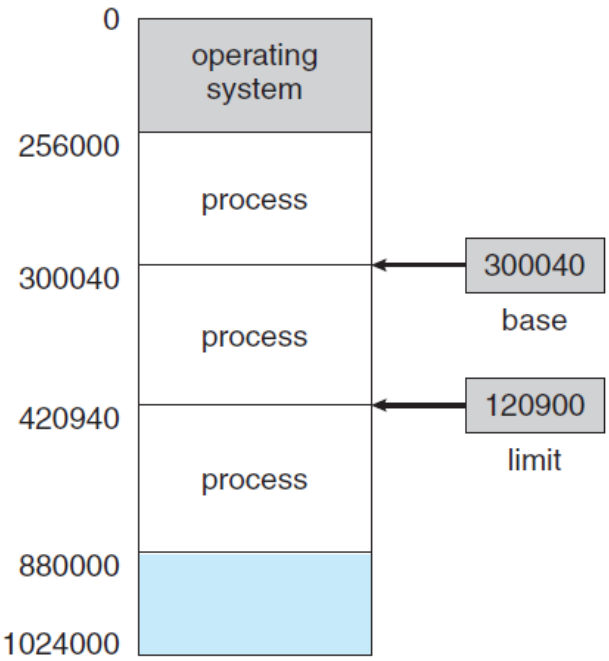

Figure 1.1: A base and a limit register define a process' logical address space

The **base register** holds the starting address; the **limit register** specifies the size of the space.

1.1 Address binding schemes

Address binding relates to how the code of a program is stored in memory. Programs are written in human-readable text, following a series of rules set up by the structural requirements of the programming language, and using keywords that are interpreted into actions by the computer's Central Processing Unit (CPU). The point at which the executable version of a program is created dictates when address binding occurs. Some program languages, such as "C" and COBOL need to be compiled, while others, mainly scripts, run from the original program text rather than a machine code compiled binary version.

An address binding can be done in three different ways:

- Compile time

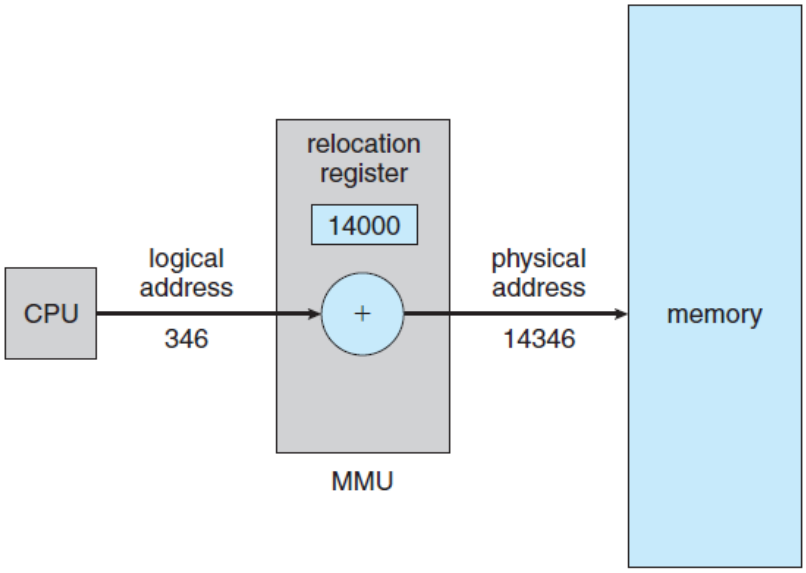

Figure 1.2: Dynamic relocation using a relocation register.

- Load time
- Execution time

1.2 Logical Versus Physical Address Space

To alleviate the task of managing memory of multiple processes, logical or virtual addresses are used. Logical addresses are independent of location in physical memory.

An address that is generated by the CPU is commonly referred to as a **logical address** or virtual 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 generates the identical logical and physical addresses. 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 known as a **logical address space**. The set of all physical addresses corresponding to these logical addresses is known as a **physical address space**. As a result, in the execution-time address-binding scheme, the logical and physical address spaces differ.

A program issues addresses in a logical address space, hence it must be translated to physical address space. To put it simply, think of a program as having a contiguous logical address space that starts at 0. That space has a corresponding contiguous physical address space that starts at somewhere else in the memory.

The run-time mapping from virtual to physical addresses is done by a hardware device called the **memory-management unit (MMU)**. Figure 1.2 illustrates an example of logical-physical address mapping using MMU. The base register is now called a relocation register. The value in the relocation register

is added to every address generated by a user process at the time it is sent to the memory. If the base is 14000, an attempt to address location 346 is dynamically relocated to location 14346 in the memory.

1.3 Dynamic loading

So far, we have only considered that the entire data and program must be in the physical memory before executing. To obtain better memory-space utilization, we can use **dynamic loading**: a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format. First, the main program is loaded into memory and is executed. Then, a routine is loaded into the memory only when it is called by another routine.

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. Therefore, swapping makes it possible that the total physical memory space of all processes can exceed the real physical memory of the system. Consequently, the degree of multi-programming in a system is augmented.

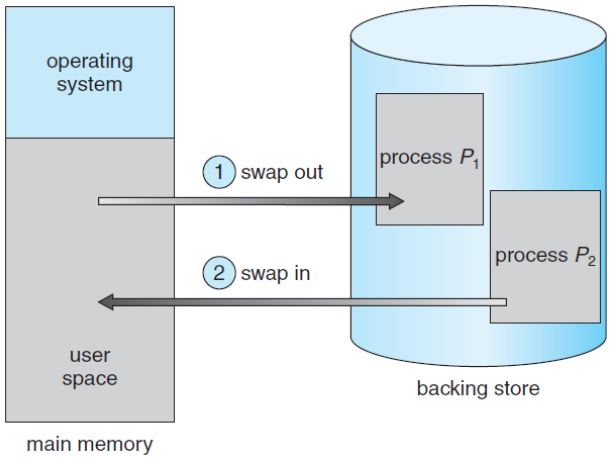

Figure 1.3: Swapping of two processes using a disk as a backing store.

Furthermore, the system maintains a ready queue of ready-to-run processes which have memory images on disk

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. The memory is usually divided into two partitions: one for the 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 Exercise

2.1 Questions

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 memory utilization.
2. Compare the advantages as well as disadvantages of these allocation algorithms: **First-Fit**, **Best-Fit**, **Worst-Fit**. Use specific examples to support your answer.

2.2 Programming exercises

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 `main.c`, we simulate a process of allocating or de-allocating memory.

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

Listing 1: main.c

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

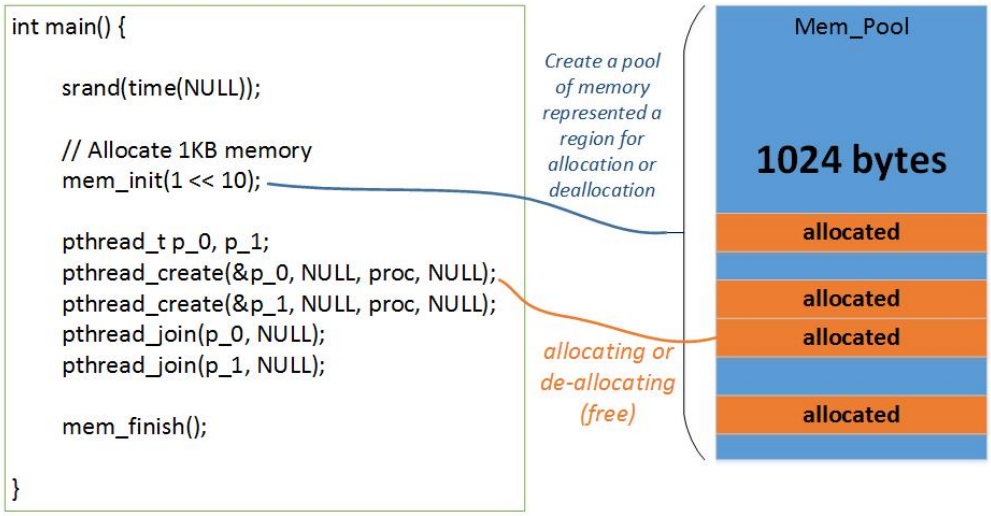

Figure 2.1: Diagram of simulating the process of allocation, deallocation in memory.

```c
#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);
    pthread_create(&p_1, NULL, proc, NULL);
    pthread_join(p_0, NULL);
    pthread_join(p_1, NULL);

    mem_finish();
}
```

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

Listing 2: mem.c

```
#include "mem.h"
#include <stdlib.h>
#include <pthread.h>
#include <stdio.h>

void * mem_pool = NULL;

pthread_mutex_t lock;

struct mem_region {
    size_t size;   // Size of memory region
    char * pointer;   // Pointer to the first byte
    struct mem_region * next; // Pointer to the next region in the list
    struct mem_region * prev; // Pointer to the previous region in the list
};

struct mem_region * free_regions = NULL;
struct mem_region * used_regions = NULL;


static void * best_fit_allocator(unsigned int size);
static void * first_fit_allocator(unsigned int size);

int mem_init(unsigned int size) {
    /* Initial lock for multi-thread allocators */
    return (mem_pool != 0);
}

void mem_finish() {
    /* Clean preallocated region */
    free(mem_pool);
}
```

```
void * mem_alloc(unsigned int size) {
   pthread_mutex_lock(&lock);
   // Follow is FIST FIT allocator used for demonstration only.
   // You need to implment your own BEST FIT allocator.
   // TODO: Comment the next line
   void * pointer = first_fit_allocator(size);
   // Commnent out the previous line and uncomment to next line
   // to invoke best fit allocator
   // TODO: uncomment the next line
   //void * pointer = best_fit_allocator(size);

   // FOR VERIFICATION ONLY. DO NOT REMOVE THESE LINES
   if (pointer != NULL) {
      printf("Alloc [%4d bytes] %p-%p\n", size, pointer, (char*)pointer + size
          - 1);
   }else{
      printf("Alloc [%4d bytes] NULL\n");
   }

   pthread_mutex_unlock(&lock);
   return pointer;
}

void mem_free(void * pointer) {
   /* free memory */
}

void * best_fit_allocator(unsigned int size) {
   // TODO: Implement your best fit allocator here
   return NULL; // remember to remove this line
}

void * first_fit_allocator(unsigned int size) {
   /* First fit example is shown in the given source code*/
}
```

2.3 Submission

Put your report in a single PDF file `report.pdf`. Move the folder `source_code` and the report into a single directory <studentID>.zip and submit to Sakai.