HCMC University Of Technology
Faculty of Computer Science & Engineering

Lab 4
Scheduling & Contiguous Memory Allocation
Course: Operating Systems

November 16, 2021

Goal: This lab helps student to practice the scheduling algorithms and the operations of contiguous memory allocation in Operating System.

Content In detail, this lab requires student implement the following tasks:

- Scheduling algorithm: use the given source code to emulate the scheduling algorithm using **FCFS** and **Round-Robin**.
- Contiguous Memory Allocation: use the given source code to implement the allocation algorithm of memory using **First-Fit**, **Best-Fit** and **Worst-Fit**.

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

Requirement Student need to review the theory of scheduling and contiguous memory allocation.

Contents

1 CPU scheduling

1.1 Workload Assumptions

Before getting into the range of possible policies, let us first make a number of simplifying assumptions about the processes running in the system, sometimes collectively called the **workload**. Determining the workload is a critical part of building scheduling policies.

We will make the following assumptions about the processes, sometimes called jobs, that are running in the system:

1. Each job runs for the same amount of time.
2. All jobs arrive at the same time.
3. All jobs only use the CPU (i.e., they perform no I/O)
4. The run-time of each job is known.

Scheduling criteria:

- CPU utilization
- Throughput
- Turnaround time: The interval from the time of submission of a process to the time of completion is the turnaround time.

$$T_{turnaround} = T_{completion} - T_{arrival} \tag{1.1}$$

- Waiting time: It affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.
- Response time: The time from the submission of a request until the first response is produced. This is called response time, is the time it takes to start responding, not the time it takes to output the response.

$$T_{response} = T_{firstrun} - T_{arrival} \tag{1.2}$$

1.2 Scheduling algorithms

1.2.1 First ŷCome, First Served (FCFS)

The most basic algorithm a scheduler can implement is known as **First In, First Out (FIFO)** scheduling or sometimes **First Come, First Served**. FIFO has a number of positive properties: it is clearly very simple and thus easy to implement. Given our assumptions, it works pretty well.

Process	Burst Time
P1	24
P2	3
P3	3

If the processes arrive in the order P1, P2, P3, and are served in FCFS order, we get the result shown in the following Gantt chart, which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes:

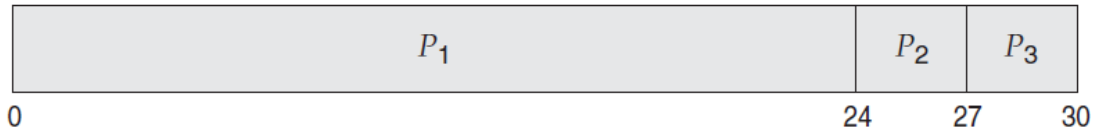

Figure 1.1: FIFO algorithm.

1.2.2 Shortest Job First Scheduling

A different approach to CPU scheduling is the **shortest-job-first** (**SJF**) scheduling algorithm. This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst.

As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

Using SJF scheduling, we would schedule these processes according to the following Gantt chart:

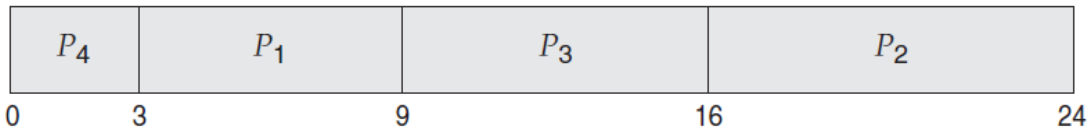

Figure 1.2: SJF algorithm.

1.2.3 Priority scheduling

The **SJF** algorithm is a special case of the general priority-scheduling algorithm. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order.

1.2.4 Round-robin scheduling

The round-robin (RR) scheduling algorithm is designed especially for time-sharing systems. It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, called **a time quantum** or **time slice**, is defined. A time quantum is generally from 10 to 100 milliseconds in length.

1.3 Practice

This lab requires student implement 2 scheduling algorithms, such as **FCFS** and **Round Robin**. You are marked at the class with the given source code and only need to add your code in "*// TO DO*" parts. The source codes emulate the scheduling process of OS using (**FCFS**) and **Round Robin**. With Round Robin, the quantum time is entered from stdin.

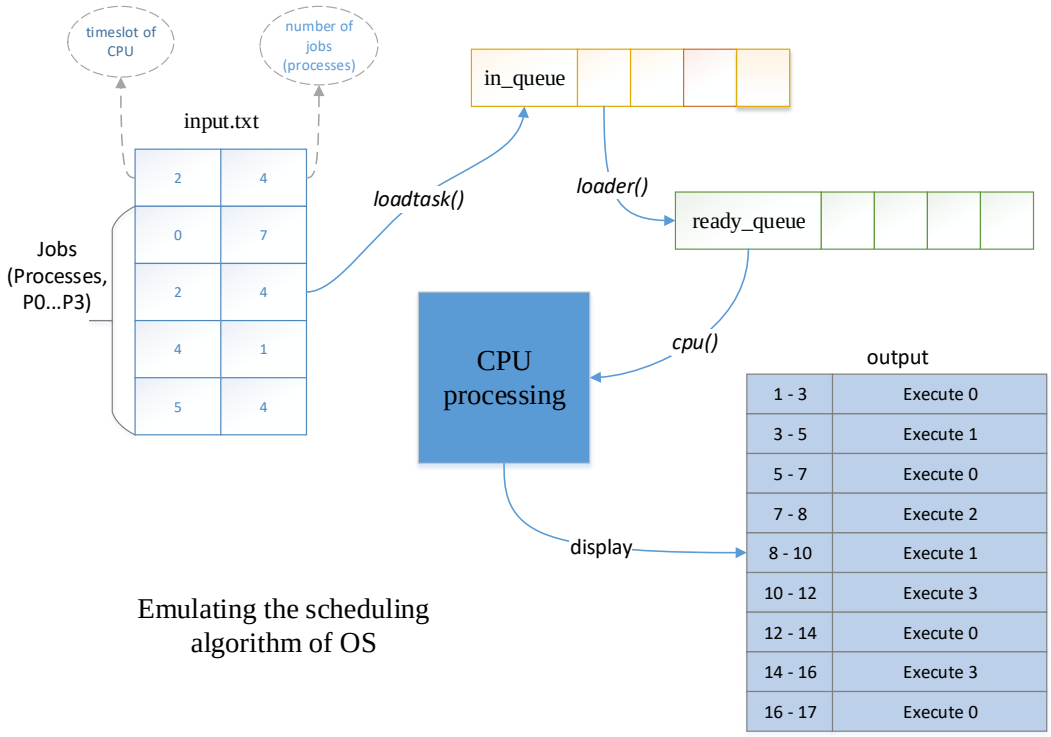

Figure 1.3: Model of emulating the scheduling algorithm in OS.

In the source code, you need to understand the content of each source file. With the `main()` function in `sched.c`, we need to do;

```
1  /////////// sched.c ////////////
2
3  #include "queue.h"
4  #include <pthread.h>
5  #include <unistd.h>
6  #include <stdio.h>
7  #include <stdlib.h>
8
9  #define TIME_UNIT        100 // In microsecond
10
11 static struct pqueue_t in_queue; // Queue for incomming processes
12 static struct pqueue_t ready_queue; // Queue for ready processes
13
14 static int load_done = 0;
15
16 static int timeslot; // The maximum amount of time a process is allowed
17 // to be run on CPU before being swapped out
18
19 // Emulate the CPU
20 void * cpu(void * arg);
21
22 // Emulate the loader
23 void * loader(void * arg);
24
25 /* Read the list of process to be executed from stdin */
26 void load_task();
27
28 int main(){
29         pthread_t cpu_id;        // CPU ID
30         pthread_t loader_id;     // LOADER ID
31
32         /* Initialize queues */
33         initialize_queue(&in_queue);
34         initialize_queue(&ready_queue);
35
36         /* Read a list of jobs to be run */
37         load_task();
38
39         /* Start cpu */
40         pthread_create(&cpu_id, NULL, cpu, NULL);
41         /* Start loader */
```

```
42          pthread_create(&loader_id, NULL, loader, NULL);
43
44          /* Wait for cpu and loader */
45          pthread_join(cpu_id, NULL);
46          pthread_join(loader_id, NULL);
47
48          pthread_exit(NULL);
49  }
```

Note: You need to consider the data structure which is declared in `structs.h`. This file declares attributes of `process` and `queue`.

```
1   /////////// structs.h /////////////
2   #ifndef STRUCTS_H
3   #define STRUCTS_H
4
5   #include <pthread.h>
6
7   /* The PCB of a process */
8   struct pcb_t {
9           /* Values initialized for each process */
10          int arrival_time; // The timestamp at which process arrives
11                            // and wishes to start
12          int burst_time; // The amount of time that process requires
13                          // to complete its job
14          int pid;        // process id
15  };
16
17  /* 'Wrapper' of PCB in a queue */
18  struct qitem_t {
19          struct pcb_t * data;
20          struct qitem_t * next;
21  };
22
23  /* The 'queue' used for both ready queue and in_queue (e.g. the list of
24   * processes that will be loaded in the future) */
25  struct pqueue_t {
26          /* HEAD and TAIL for queue */
27          struct qitem_t * head;
28          struct qitem_t * tail;
29          /* MUTEX used to protect the queue from
30           * being modified by multiple threads*/
31          pthread_mutex_t lock;
32  };
```

```
33
34 #endif
```

Finally, that is the source file used to implement the `queue` operations, such as `en_queue()`, `de_queue()`.

```
1  /////////// queue.h ////////////
2
3  #ifndef QUEUE_H
4  #define QUEUE_H
5
6  #include "structs.h"
7
8  /* Initialize the process queue */
9  void initialize_queue(struct pqueue_t * q);
10
11 /* Get a process from a queue */
12 struct pcb_t * de_queue(struct pqueue_t * q);
13
14 /* Put a process into a queue */
15 void en_queue(struct pqueue_t * q, struct pcb_t * proc);
16
17 int empty(struct pqueue_t * q);
18
19 #endif
```

2 Contiguous Memory Allocation

2.1 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 2.1 shown. The

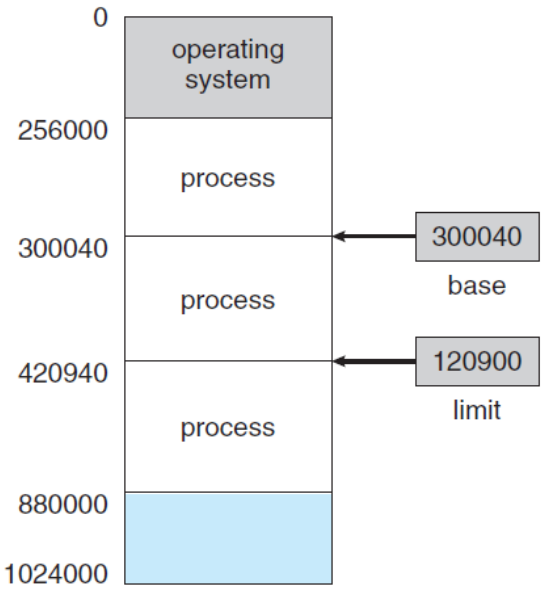

Figure 2.1: A base and a limit register define a logical address space.

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

2.1.1 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:

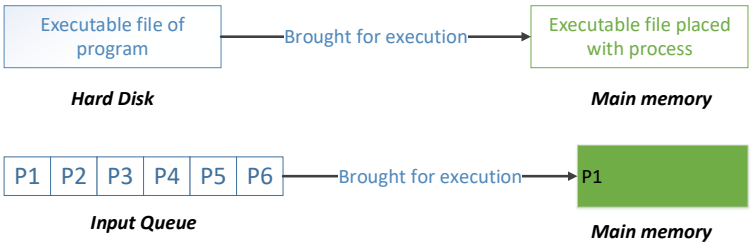

Figure 2.2: Process moves between disk and memory during execution.

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

2.1.2 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.

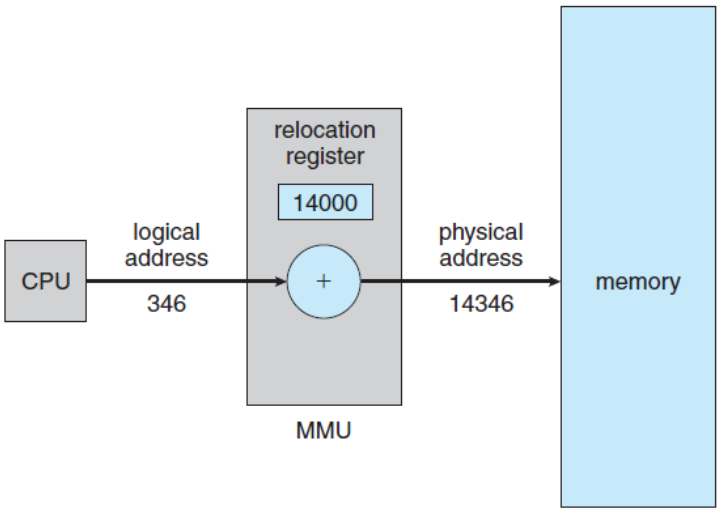

Figure 2.3: Dynamic relocation using a relocation register.

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)**.

2.1.3 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.

2.1.4 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.

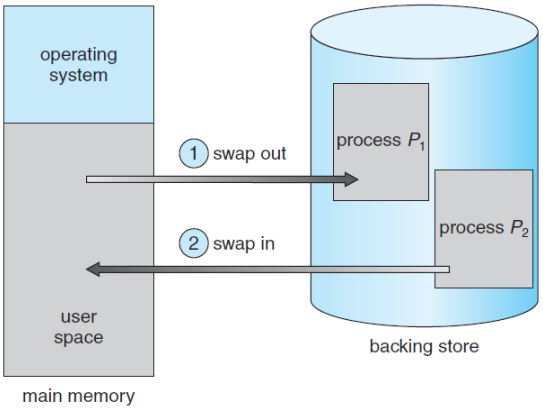

Figure 2.4: Swapping of two processes using a disk as a backing store.

2.1.5 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.

2.2 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.

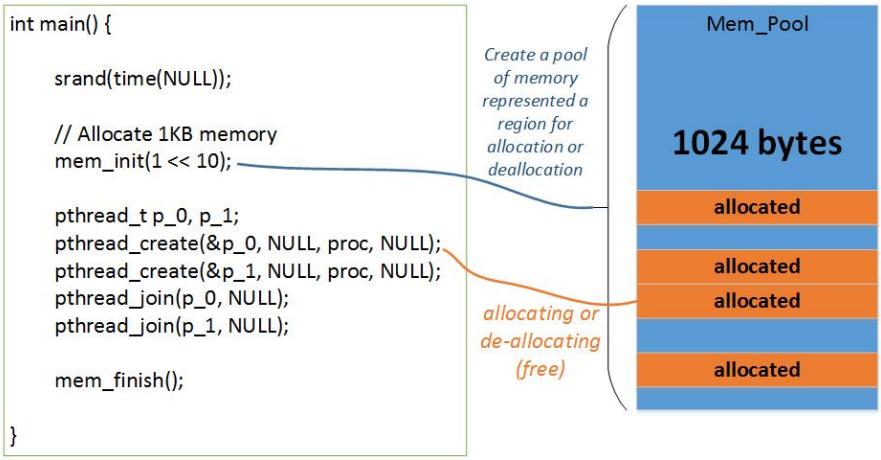

Figure 2.5: Diagram of simulating the process of allocation, deallocation in memory.

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** and **Worst-fit** allocators.

```
1  ////////// main.c ////////////
2  #include "mem.h"
3  #include <stdio.h>
4  #include <pthread.h>
5  #include <stdlib.h>
6  #include <time.h>
7
8  #define ARRAY_SIZE 10
9
10 void * proc(void *args) {
11         int i;
12         int index = 0;
13         char * mem[ARRAY_SIZE];
14         for (i = 0; i < ARRAY_SIZE; i++) {
15                 if (rand() % 2) {
16                         /* Allocate memory */
17                         unsigned int size = 1 << ((rand() % 4) + 4);
18                         mem[index] = mem_alloc(size);
19                         if (mem[index] != NULL) {
20                                 index++;
21                         }
22                 }else{
23                         /* Free memory */
24                         if (index == 0) {
25                                 continue;
26                         }
27                         unsigned char j = rand() % index;
28                         mem_free(mem[j]);
29                         mem[j] = mem[index - 1];
30                         index--;
31                 }
32         }
33 }
34
35 int main() {
36
```

```
37          srand(time(NULL));
38
39          // Allocate 1KB memory pool
40          mem_init(1 << 10);
41
42          pthread_t p_0, p_1;
43          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 }
```

3 Exercises

1. Write two programs for CPU scheduling using **FCFS** and **Round Robin**. Measure the total turnaround time for all processes.(read the section 1.3)

2. Write two programs for the Contiguous Memory Allocation using algorithm **Best-fit** and **Worst-fit** by completing the TODO part in the sample code. (read the section 2.2)