SCS1201 - Data Structures & Algorithms I

Assignment - 5

Student Name : Ramindu Walgama

Student Temp ID : 201177

Submission Date : 28. 08. 2021

Table of Contents

1. Introduction

This is a program to investigate the process of the first fit memory management and sleeping barber problem. First fit is the memory management algorithm used in computers. The chapter is regarding the implementation of this algorithm. The second chapter is about the common concurrent problem called the sleeping barber problem. This question is completed without using the concurrent programming but using an alternative algorithm.

2. First-fit Memory Management

2.1 Implementation

First, implement the linked list and took each node as a process.
It consists of the following methods.

Function	Return	Parameters
`void firstFit(struct Node *head_, char *process_id__, int size_)`		
Insert a new process into the memory according to the first-fit algorithm. (Insert a node to the linked list)	void	• head_ - head of the linked list • process_id_ - process id : example : p1, p2 • start_ - starting memory location of the process in the memory example: p1 starts after the OS allocation. Start will be the end of the allocated memory for OS. If OS is 400K, start of the p1 (only process) will be 400k. • size_ - size of the process.
`void showTasks(struct Node *head_)`		
Print a snapshot of the memory	void Print a snapshot of the memory	• head_ - head of the linked list
`struct Node* getProcess(struct Node *head_, char *process_id_)`		

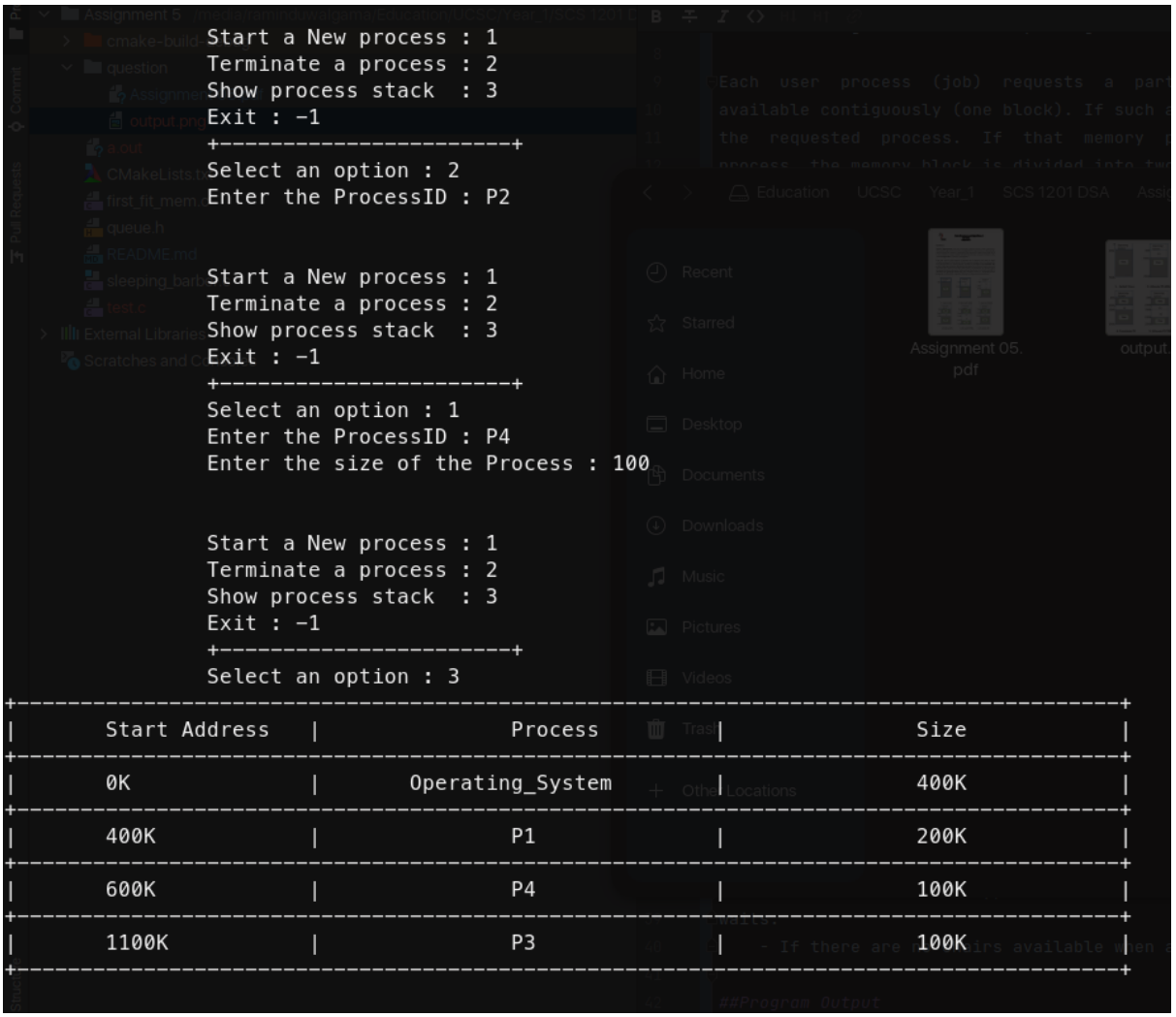
Check a given process (node) exists	Given process if a process exists. (a node) Return type: Node pointer	• head_ - head of the linked list • process_id_ - process id : example : p1, p2
`void removeProcess(struct Node *head_, char* process_id_)`		
Remove a process (Delete a node from the linked list)	void	• head_ - head of the linked list • process_id_ - process id : example : p1, p2
`int main()`		
Main thread: handles user inputs and prompts and manages the user interface.	-	-
`void printb(char* string_)`		
Method to bold text, errors	void Bold text output in the terminal	string _ - the text which wants to bold

2.2 Data structure

Used doubly-linked list to maintain the process. First thought of a single linked list and completed up to a certain point in the program but when wanted to go back to the previous node, required the double linked list. The reason why wanted to iterate to the previous node is that when want to find an element(whether exists in the linked list), used a method called *getProcess*. This method will find the node and return the node (without removing the node).

This is being used in two other places in the code; Deleting a node and when adding a new element to the linked list to check whether the node exists. If exists won't let the user add that node(process). The problem occurred when wanted to delete a node from the linked list. Wanted to return the previous node if using a single node or else use a doubly-linked list so can find the previous node simply and connect with correct pointers. The reason why give up on the first node method is felt that was not a good method and this has perfect flow.

2.3 Sample output- valid test data and results



```
Start a New process : 1
Terminate a process : 2
Show process stack  : 3
Exit : -1
+-----------------------+
Select an option : 2
Enter the ProcessID : P2


Start a New process : 1
Terminate a process : 2
Show process stack  : 3
Exit : -1
+-----------------------+
Select an option : 1
Enter the ProcessID : P4
Enter the size of the Process : 100


Start a New process : 1
Terminate a process : 2
Show process stack  : 3
Exit : -1
+-----------------------+
Select an option : 3
+-------------------------------------------------------------------------------+
|       Start Address   |              Process          |              Size     |
+-------------------------------------------------------------------------------+
|       0K              |       Operating_System        |              400K     |
+-------------------------------------------------------------------------------+
|       400K            |              P1               |              200K     |
+-------------------------------------------------------------------------------+
|       600K            |              P4               |              100K     |
+-------------------------------------------------------------------------------+
|       1100K           |              P3               |              100K     |
+-------------------------------------------------------------------------------+
```

3. Sleeping Barber

3.1 Implementation

First, implement the queue, and an element of the queue is a customer (string).
Queue consists of the following methods.

Function	Return	Parameters
`int isFull()`		
to check the queue is full	boolean - 1 or 0	-
`int isEmpty()`		
to check queue is empty	boolean - 1 or 0	-
`void enQueue(char* name_)`		
Add an element to the queue	void	name_ - string (Customer name)
`char* deQueue()`		
remove an element from the queue	char* deQueue()	-
`void displaySeats()`		
display a snapshot of the queue (chairs)	void - print the queue	-
`void printb(char* string_)`		
Method to bold text, errors	void Bold text output in the terminal	string _ - the text which wants to bold

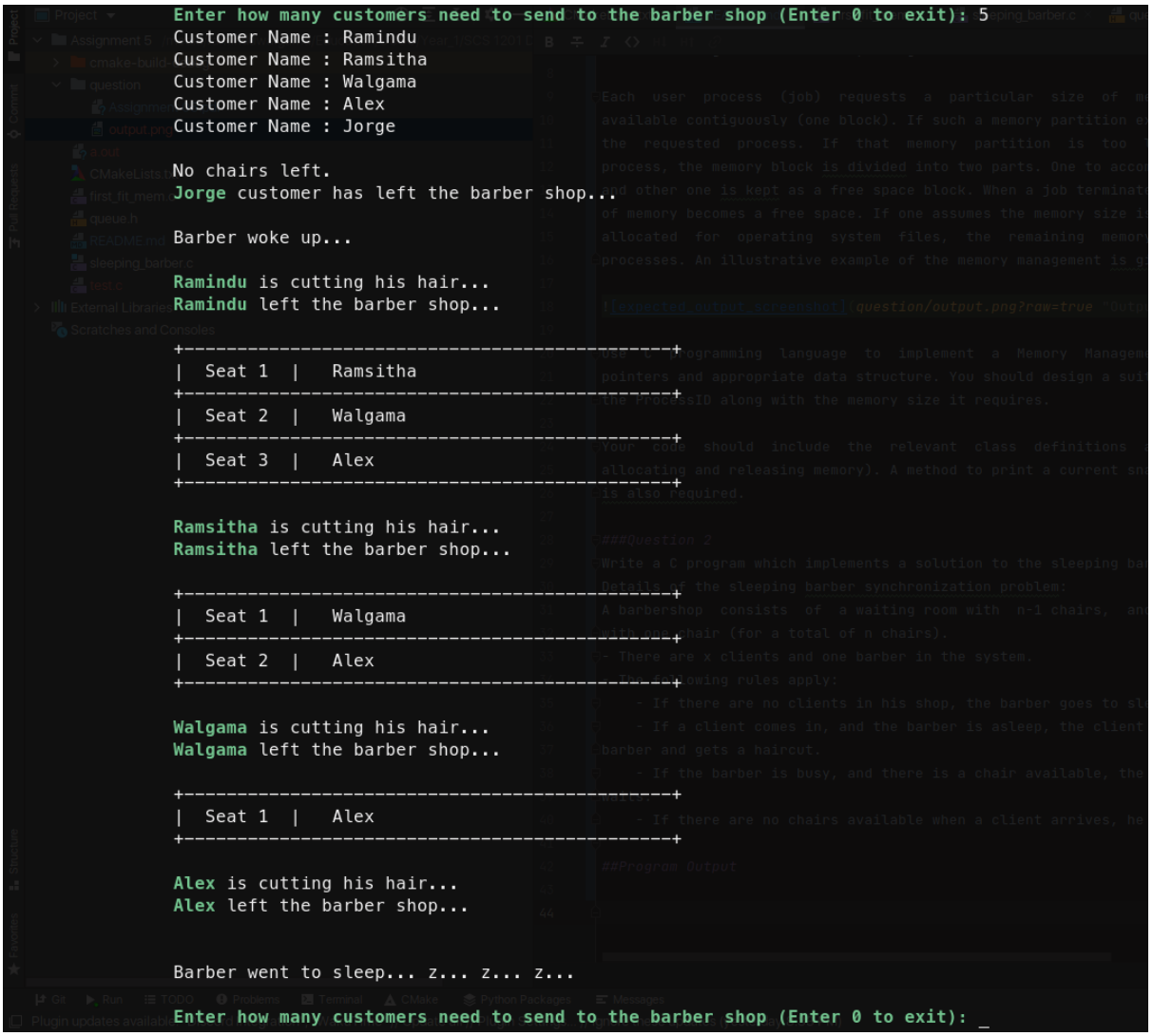
In the main method, did the following process inside a do while loop till the user input is 0. Get the user input for the number of customers needed to send to the barber shop. If it's 0 exists from the do while loop and end the loop. If user inputs a negative value for the customer count, it will prompt and error and ask for the count again. Then user redirect to

input set of names of the customers. If a user enters a count more than the seats available in the barber shop, the last customer which was taken by the user, who doesn't have a seat will leave the shop. Then the customers in the queue will go to cut the hair in order and the barber will take a random time to cut each customer's hair. When he cuts the hair of all of the customers, he goes to sleep till the user inputs another customer list.

Important:

Question is requested to implement a synchronization method since C doesn’t not support concurrent programming, went through the above implementation. I have gone through multiple multithreading libraries but the problem is that the program won't run without those dependencies in another computer. This is the reason why I implemented this problem in the above algorithm.

3.2 Sample output - valid test data and results



4. Code

4.1 First-fit mem management algorithm

first_fit_mem.c

```c
#include <stdio.h>
#include <malloc.h>
#include <string.h>

/*
 * Name     : Ramindu Walgama
 * Index    : 201177
 * Email    : raminduw200@gmail.com
 */

#define MEMORY 2560
int remaining_mem = 2560;

void printb(char* string_);

// Node = Process/ Job
struct Node {
      char *process_id;
      int start;
      int size;
      struct Node *next;
      struct Node *prev;
};

/*
 * @function: Insert a new process into the memory according to
the first-fit
 *          algorithm. (Insert a node to the linked list)
 * @return   : void
 * @args     : head_     - head of the linked list
 *          process_id_ - process id : example : p1, p2
 *          start_      - starting memory location of the process in
the memory
 *                        example: p1 starts after the OS
```

```
allocation. Start will be the end of the allocated
 *                          memory for OS. If OS is 400K, start of
the p1 (only process) will be 400k.
 *         size_       - size of the process.
 */
void firstFit(struct Node *head_, char *process_id__, int size_) {
     char *process_id_ = malloc(strlen(process_id__) + 1);
     strcpy(process_id_, process_id__);

//   Check the memory is full or not.
     if (remaining_mem - size_ >= 0) {
     struct Node *temp = head_;
     struct Node *new_node = (struct Node *) malloc(sizeof(struct
Node));

     new_node->process_id = process_id_;
     new_node->size = size_;
     new_node->next = NULL;

/*   Check place where the new_node should be(where the process
should be in the memory)
 *   let new_node = P2 and goes between P1 and P3
 *   OS ------> P1 ------> P3
 *   while loop assign the P1 to the temp since its the place
where new node belongs
 *                               4.temp -> next = new_node
     1.new_node -> next = temp -> next
 *                               |--------------------------->|
     |--------------------------->|
 *   OS -------------> P1 (temp)                          P2
(new_node)                          P3
 *                               |<---------------------------|
     |<---------------------------|
 *                               2.new_node -> prev = temp
     3.temp -> next -> prev = new_node
 */
     while (temp->next != NULL) {
          if (temp->next->start - (temp->start + temp->size) >=
size_) {
               new_node->start = temp->start + temp->size;
               new_node->next = temp->next;
               new_node->prev = temp;
```

```
                temp->next->prev = new_node;
                temp->next = new_node;
                remaining_mem -= size_;
                break;
            }
            temp = temp->next;
        }

//          If only the OS exists as a process in the memory, add
new process next to OS.
        if (head_->next == NULL) {
            new_node->start = head_->start + head_->size;
            head_->next = new_node;
            new_node->prev = head_;
            remaining_mem -= size_;
        }
//          If the available memory is in the last space, add new
process to the last
        else if (MEMORY - (temp->start + temp->size) > size_) {
            new_node->start = temp->start + temp->size;
            temp->next = new_node;
            new_node->prev = temp;
        }
        else {
            printb("\n\t\t[ERROR] Ran out of memory.\n");
        }
        } else {
        printb("\n\t\t[ERROR] Ran out of memory.\n");
        }
}


/*
 * @function : Print a snapshot of the memory
 * @return   : void
 *               print the snapshot
 * @args     : head_     - head of the linked list
 */
void showTasks(struct Node *head_) {
        struct Node *temp = head_;

printf("+-------------------------------------------------------
```

```
------------------------------+\n");
      printb("|\tStart
Address\t|\t\tProcess\t\t|\t\tSize\t\t|\t\t\n");

printf("+-------------------------------------------------------------
------------------------------+\n");
      printf("|\t%dK\t\t|\t%s\t|\t\t%dK\t\t|\t\t\n", head_->start,
head_->process_id, head_->size);

printf("+-------------------------------------------------------------
------------------------------+\n");

      while (temp->next != NULL) {
      temp = temp->next;
      printf("|\t%dK\t\t|\t\t%s\t\t|\t\t%dK\t\t|\t\t\n",
temp->start, temp->process_id, temp->size);

printf("+-------------------------------------------------------------
------------------------------+\n");
      }
}


/*
 * @function : Check a given process (node) exists
 * @return   : Given process if process exists. (a node)
 *               @type: Node pointer
 * @args    : head_     - head of the linked list
 *               process_id_ - process id : example : p1, p2
 */
struct Node* getProcess(struct Node *head_, char *process_id_){
      struct Node* temp = head_;

      if(head_->process_id == process_id_)
      return head_;

      while (temp->next != NULL){
      temp = temp->next;
      if(strcmp(temp->process_id, process_id_) == 0) //
temp->process_id == process_id_
            return temp;
      }
```

```
	return NULL;
}


/*
 * @function : Remove a process (Delete a node from the linked
list)
 * @return   : void
 * @args     : head_     - head of the linked list
 *             process_id_ - process id : example : p1, p2
 */
void removeProcess(struct Node *head_, char* process_id_) {
	struct Node* deleteNode = getProcess(head_, process_id_);

	if (deleteNode == head_){
	printb("\n\t\t[ERROR] Cannot terminate the Operating
System.\n");
	} else if (deleteNode->next !=NULL){
	deleteNode->next->prev = deleteNode->prev;
	deleteNode->prev->next = deleteNode->next;
	free(deleteNode);
	} else if(deleteNode->next == NULL){
	deleteNode->prev->next = NULL;
	free(deleteNode);
	}
}


// main thread
int main() {
	int option;
	int process_size;

	//    Head will be the OS.
	struct Node head;
	head.process_id = "Operating_System";
	head.start = 0;
	head.size = 400;
	head.next = NULL;
	head.prev = NULL;
```

```
printf("\n+-------------------------------------------------------------
--------------------------------+\n");
     printb("+-----------------------------First Fit Memory
Management-------------------------------+\n");

printf("+-------------------------------------------------------------
-----------------------------+\n");

     do {
     char process_id[20];

     printf("\n\n\t\tStart a New process : 1\n");
     printf("\t\tTerminate a process : 2\n");
     printf("\t\tShow process stack  : 3\n");
     printf("\t\tExit : -1\n");
     printf("\t\t+----------------------+\n");
     printf("\t\tSelect an option : ");
     scanf("%d", &option);


     switch (option) {
          case 1: // Start a New process
               printf("\t\tEnter the ProcessID : ");
               scanf("%s", process_id);
//             check the process is already running
               if (getProcess(&head, process_id) == NULL) {
               printf("\t\tEnter the size of the Process : ");
               scanf("%d", &process_size);
               firstFit(&head, process_id, process_size);
               } else {
               printb("\n\t\t[WARNING] Process is already
running.\n");
               }
               break;
          case 2: // Terminate a process
               printf("\t\tEnter the ProcessID : ");
               scanf("%s", process_id);
               removeProcess(&head, process_id);
               break;
          case 3: // Show process stack
               showTasks(&head);
```

```
                break;
            default:
                printb("\t\tPlease enter a valid input\n");
                break;
        }
        } while (option != -1);

        return 0;
}

// Method to bold text, errors
void printb(char* string_){
        printf("\e[1m%s\e[0m", string_);
};
```

queue.h

```
#include <string.h>
#include <stdio.h>

#define SIZE 4

int front = -1;
int rear = -1;
char queue[SIZE][50];
void printb(char* string_);

/*
 * @function : to check the queue is full
 * @return   : boolean - 1 or 0
 * @args     : -
 */
int isFull(){
      if(rear + 1 == front || (rear == SIZE-1 && front == 0))
      return 1;
      return 0;
}

/*
 * @function : to check queue is empty
 * @return   : boolean - 1 or 0
 * @args     : -
 */
int isEmpty(){
      if(front == -1)
      return 1;
      return 0;
}

/*
 * @function : Add an element to the queue
 * @return   : void
 * @args     : name_ - string
 */
void enQueue(char* name_){
      if (isFull()){
      printb("\t\t[ERROR] Queue is Full. Error occurs while
enqueueing new element");
```

```
      printf("%s\n", name_);
      } else {
      if (front == -1) ++front;
      rear = (rear+1)%SIZE;
      strcpy(queue[rear], name_);
      }
}

/*
 * @function : remove an element from the queue
 * @return   : string - removed value from the queue
 * @args     : -
 */
char* deQueue(){
      int temp;
      if (isEmpty()){
      printb("\t\t[ERROR] Queue is Empty.Error occurs while
de-queueing the empty queue.\n");
      return NULL;
      } else {
      temp = front;
      if (rear == front) rear = front = -1;
      else front = (front + 1) % SIZE;
      return queue[temp];
      }
}


/*
 * @function : display a snapshot of the queue (chairs)
 * @return   : void - print the queue
 * @args     : -
 */
void displaySeats(){
      int count=0;
      if (isEmpty()){
      printb("\t\t[ERROR] Queue is Full. Error occurs while
printing the queue\n");
      } else {
      for (int i = front; i < rear+1; ++i) {

printf("\t\t+-----------------------------------------------+\n");
```

```
            printf("\t\t|  Seat %d  |   %s\n", ++count, queue[i]);
      }

printf("\t\t+-----------------------------------------------+\n");
      }
}

// Method to bold text, errors
void printb(char* string_){
      printf("\e[1m%s\e[0m", string_);
};
```

sleeping_barber.c

```
#include "queue.h"
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

/*
 * Name    : Ramindu Walgama
 * Index   : 201177
 * Email   : raminduw200@gmail.com
 */

int main(){
      int cust_count;
      char* name;

      do {
      printb("\t\tEnter how many customers need to send to the
barber shop (Enter 0 to exit): ");
      scanf("%d", &cust_count);

      if (cust_count < 0){
            printb("[ERROR] Customer count can not be a negative
value.");
            continue;
      }

      char cust_list[cust_count][100];

      // input all customer names
      for (int i = 0; i < cust_count; ++i) {
            printf("\t\tCustomer Name : ");
            scanf("%s", cust_list[i]);
      }

      // Enqueue
      for (int i = 0; i < cust_count; ++i) {
            name = cust_list[i];
            if (isFull()) {
                  printf("\n\t\tNo chairs left.\n\t\t");
```

```
                printb(name);
                printf(" customer has left the barber shop...\n");
          }
          else
                enQueue(name);
     }

     printf("\n\t\tBarber woke up...\n\n");

     while (!isEmpty()) {
          char* cust = deQueue();

          printf("\t\t");
          printb(cust);
          printf(" is cutting his hair...\n");

          // Sleep - Random number between 7 and 3
          sleep(rand() % (7+1-3) + 3);

          printf("\t\t");
          printb(cust);
          printf(" left the barber shop...\n\n");

          if (!isEmpty())
                displaySeats();
          printf("\n");
     }

     printf("\t\tBarber went to sleep... z... z... z...\n\n");

     } while (cust_count != 0);
     return 0;
}
```