Experiment-1

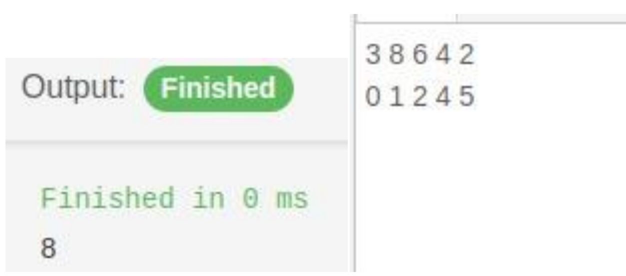
Aim: Write a program to implement CPU Scheduling for First Come Serve.

Theory:

Source Code:

```
void fcfs(vector<int> burst_time,vector<int> arrival_time)
{
    float wt=0;
    for(int i=0;i<burst_time.size();i++)
    {
        wt+=burst_time[i];
        burst_time[i]=wt;
    }
    wt=0;
    for(int i=0;i<burst_time.size()-1;i++)
    {
       wt+=(burst_time[i]-arrival_time[i+1]);
    }
    cout<<wt/burst_time.size();
    return ;
}
int main() {
    vector<int> b_t;
    vector<int> a_t;
    for(int i=0;i<5;i++)
    {
        int element;
        cin>>element;
        b_t.push_back(element);
    } for(int i=0;i<5;i++)
    {
        int element;
        cin>>element;
        a_t.push_back(element);
    }
fcfs(b_t,a_t);
    return 0;
}
```

Output:



Experiment-2 A

Aim: Write a program to implement CPU Scheduling for Shortest Job First for preemptive.

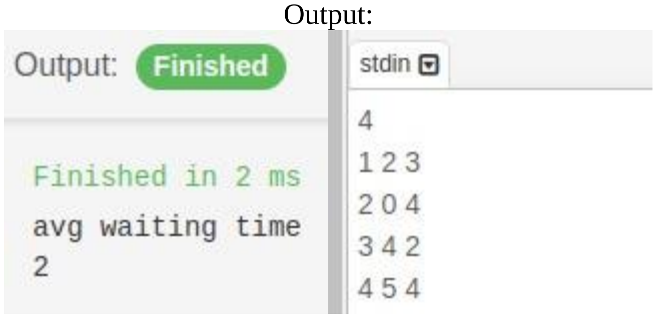
Theory:

Source Code:

```
bool comp(vector<int> a,vector<int> b)
{
    if(a[1]==b[1])
    {
        if(a[2]==b[2])
        {
            return a[0]<b[0];

        }
        else
        {
            return a[2]<b[2];
        }
    }
    return a[1]<b[1];
}
int main() {
    int n;
    cin>>n;
    vector<vector<int>> v(n);
    for(int i=0;i<n;i++)
    {
        for(int j=0;j<3;j++)
        {
            int x;
            cin>>x;
            v[i].push_back(x);
        }
    }
    vector<int> ans;
    sort(v.begin(),v.end(),comp);
    priority_queue<vector<int>,vector<vector<int>>,greater<vector<int>>> pq;
    pq.push({v[0][2],v[0][1],v[0][0]});
    int i=1;
    int current_time=0;
    int wt=0;
    int gc=0;
    while(!pq.empty())
    {
        int burst_time=pq.top()[0];
        int id=pq.top()[2];
        int arrival_time=pq.top()[1];
        pq.pop();
        ans.push_back(id);
        if(i>1)
        {
            wt+=(current_time-arrival_time);

        }
```

```
        current_time=current_time+burst_time;

        while(true)
        {
            if(i<n and v[i][1] <= current_time)
            {
                pq.push({v[i][2],v[i][1],v[i][0]});
                i++;
            }
            else
            {
                break;
            }
        }
    }
    cout<<double(wt)/n;
    return 0;
}
```

Output:



Experiment-2B

Aim: Write a program to implement CPU Scheduling for Shortest Job First for non-preemptive.

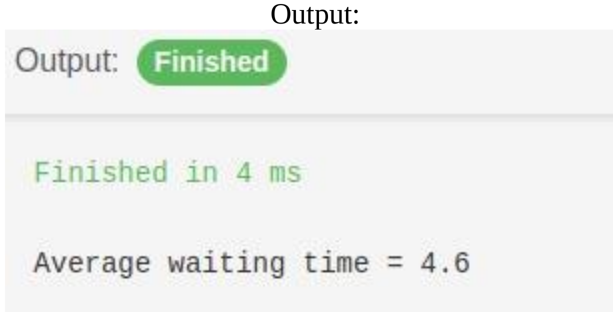
Theory:

Source Code:

```
#include <bits/stdc++.h>
using namespace std;

struct Process {
	int pid;
	int bt;
	int art;
};

void findWaitingTime(Process proc[], int n,int wt[])
{
	int rt[n];

	for (int i = 0; i < n; i++)
		rt[i] = proc[i].bt;

	int complete = 0, t = 0, minm = INT_MAX;
	int shortest = 0, finish_time;
	bool check = false;
	while (complete != n) {

		for (int j = 0; j < n; j++) {
			if ((proc[j].art <= t) &&
			(rt[j] < minm) && rt[j] > 0) {
				minm = rt[j];
				shortest = j;
				check = true;
			}
		}

		if (check == false) {
			t++;
			continue;
		}

		rt[shortest]--;

		minm = rt[shortest];
		if (minm == 0)
			minm = INT_MAX;


		if (rt[shortest] == 0) {

			complete++;
			check = false;

			finish_time = t + 1;

			wt[shortest] = finish_time -
```

```
                                                proc[shortest].bt -
                                                proc[shortest].art;

                              if (wt[shortest] < 0)
                                    wt[shortest] = 0;
                        }
                        t++;
            }
}

void findTurnAroundTime(Process proc[], int n, int wt[], int tat[])
{

            for (int i = 0; i < n; i++)
                        tat[i] = proc[i].bt + wt[i];
}

void findavgTime(Process proc[], int n)
{
            int wt[n], tat[n], total_wt = 0,total_tat = 0;
            findWaitingTime(proc, n, wt);

            findTurnAroundTime(proc, n, wt, tat);


            for (int i = 0; i < n; i++) {
                        total_wt = total_wt + wt[i];

            }

            cout << "\nAverage waiting time = "
                        << (float)total_wt / (float)n;

}
int main()
{
            Process proc[] = { { 1, 6, 2 }, { 2, 2, 5 },{ 3, 8, 1 }, { 4, 3, 0}, {5, 4, 4} };
            int n = sizeof(proc) / sizeof(proc[0]);
            findavgTime(proc, n);
            return 0;
}
```

Output:



Experiment-3

Aim: Write a program to perform priority Scheduling.

Theory:

Source Code:

```
#include <bits/stdc++.h>
using namespace std;

struct Process {
	int pid;
	int bt;
	int priority;
};


bool comparison(Process a, Process b)
{
	return (a.priority > b.priority);
}
void findWaitingTime(Process proc[], int n, int wt[])
{
	// waiting time for first process is 0
	wt[0] = 0;

	// calculating waiting time
	for (int i = 1; i < n; i++)
		wt[i] = proc[i - 1].bt + wt[i - 1];
}

void findTurnAroundTime(Process proc[], int n, int wt[],
								int tat[])
{

	for (int i = 0; i < n; i++)
		tat[i] = proc[i].bt + wt[i];
}

void findavgTime(Process proc[], int n)
{
	int wt[n], tat[n], total_wt = 0, total_tat = 0;

	findWaitingTime(proc, n, wt);

	findTurnAroundTime(proc, n, wt, tat);


	for (int i = 0; i < n; i++) {
		total_wt = total_wt + wt[i];
		total_tat = total_tat + tat[i];

	}

	cout << "\nAverage waiting time = "
		<< (float)total_wt / (float)n;
```

```
}

void priorityScheduling(Process proc[], int n)
{
	sort(proc, proc + n, comparison);

	cout << "Order in which processes gets executed \n";
	for (int i = 0; i < n; i++)
		cout << proc[i].pid << " ";

	findavgTime(proc, n);
}

int main()
{
	Process proc[]
		= { { 1, 10, 2 }, { 2, 5, 0 }, { 3, 8, 1 } };
	int n = sizeof proc / sizeof proc[0];
	priorityScheduling(proc, n);
	return 0;
}
```

Output

```
Order in which processes gets executed
1 3 2
Average waiting time = 9.33333
```

Experiment - 4

Aim: Write a program to implement CPU scheduling for Round Robin

Theory:

Source Code:

```
#include<iostream>
using namespace std;
void findWaitingTime(int processes[], int n,
                        int bt[], int wt[], int quantum)
{
        int rem_bt[n];
        for (int i = 0 ; i < n ; i++)
                rem_bt[i] = bt[i];

        int t = 0;

        while (1)
        {
                bool done = true;

                for (int i = 0 ; i < n; i++)
                {
                        if (rem_bt[i] > 0)
                        {
                                done = false;

                                if (rem_bt[i] > quantum)
                                {
                                        t += quantum;
                                        rem_bt[i] -= quantum;
                                }

                                else
                                {
                                        t = t + rem_bt[i];
                                        wt[i] = t - bt[i];
                                        rem_bt[i] = 0;
                                }
                        }
                }

                if (done == true)
                break;
        }
}



void findavgTime(int processes[], int n, int bt[], int quantum)
{
```

```
	int wt[n], tat[n], total_wt = 0, total_tat = 0;

	findWaitingTime(processes, n, bt, wt, quantum);

	for (int i=0; i<n; i++)
	{
		total_wt = total_wt + wt[i];

	}

	cout << "Average waiting time = "
		<< (float)total_wt / (float)n;

}


int main()
{

	int processes[] = { 1, 2, 3};
	int n = sizeof processes / sizeof processes[0];
	int burst_time[] = {10, 5, 8};
	int quantum = 2;
	findavgTime(processes, n, burst_time, quantum);
	return 0;
}
```

Output:

Output: Finished

```
Finished in 0 ms
Average waiting time = 12
```

Experiment- 5A

Aim: Write a program for page replacement policy using LRU.

Theory:

Source Code:

```
#include <iostream>
#include <list>
#include <unordered_map>

class LRUCache {
private:
    int capacity;
    std::list<int> order;
    std::unordered_map<int, std::list<int>::iterator> cache;

public:
    LRUCache(int capacity) : capacity(capacity) {}

    void accessPage(int page) {
        auto it = cache.find(page);
        if (it != cache.end()) {
            order.erase(it->second);
        } else {
            if (order.size() >= capacity) {
                int lruPage = order.front();
                order.pop_front();
                cache.erase(lruPage);
            }
        }

        order.push_back(page);
        cache[page] = --order.end();
    }

    void displayCache() {
        std::cout << "Current Cache: ";
        for (int page : order) {
            std::cout << page << " ";
        }
        std::cout << std::endl;
    }
};

int main() {
    int capacity;
    std::cout << "Enter the capacity of the cache: ";
    std::cin >> capacity;

    LRUCache lruCache(capacity);

    int pages[] = {1, 2, 3, 1, 4, 5};
    for (int page : pages) {
        lruCache.accessPage(page);
        lruCache.displayCache();
```

```
    }

    return 0;
}
```

Output:

```
Enter the capacity of the cache: 3
Current Cache: 1
Current Cache: 1 2
Current Cache: 2 3 1
Current Cache: 3 1 4
Current Cache: 1 4 5
```

Experiment-5B

Aim: Write a program for page replacement policy using FIFO.

Theory:

Source Code:

```
#include <iostream>
#include <list>
#include <unordered_set>
using namespace std;

class FCFSCache {
private:
    int capacity;
    list<int> order;  // Represents the order in which pages are accessed
    unordered_set<int> cache;  // Represents the set of pages in the cache
    int pageFaultCount;  // Counter for page faults

public:
    FCFSCache(int capacity) : capacity(capacity), pageFaultCount(0) {}

    void accessPage(int page) {
        // If the page is not in the cache
        if (cache.find(page) == cache.end()) {
            // If the cache is full, remove the oldest page
            if (order.size() >= capacity) {
                int oldestPage = order.front();
                order.pop_front();
                cache.erase(oldestPage);
            }

            // Add the new page to the cache and update the order
            order.push_back(page);
            cache.insert(page);

            // Increment the page fault count
            pageFaultCount++;
        }
    }

    void displayCache() {
        cout << "Current Cache: ";
        for (int page : order) {
            cout << page << " ";
        }
        cout << "\tPage Faults: " << pageFaultCount << endl;
    }
};

int main() {
    int capacity;
    cout << "Enter the capacity of the cache: ";
    cin >> capacity;

    FCFSCache fcfsCache(capacity);
```

```
    int pages[] = {1, 2, 3, 1, 4, 5};

    // Access pages
    for (int page : pages) {
        fcfsCache.accessPage(page);
        fcfsCache.displayCache();
    }

    return 0;
}
```

Output:

```
Current Cache: 1        Page Faults: 1
Current Cache: 1 2      Page Faults: 2
Current Cache: 1 2 3    Page Faults: 3
Current Cache: 1 2 3    Page Faults: 3
Current Cache: 2 3 4    Page Faults: 4
Current Cache: 3 4 5    Page Faults: 5
```

Experiment- 5C

Aim: Write a program for page replacement policy using Optimal.

Theory:

Source Code:

```
#include <iostream>

#include <list>
#include <unordered_set>
#include <vector>
#include <climits>
#include <algorithm>
using namespace std;

class OptimalCache {
private:
  int capacity;
  list<int> order;  // Represents the order in which pages are accessed
  unordered_set<int> cache;
  vector<int> futureAccesses;
  int pageFaultCount;

public:
  OptimalCache(int capacity, const vector<int>& futureAccesses)
    : capacity(capacity), futureAccesses(futureAccesses), pageFaultCount(0) {}

  void accessPage(int page) {
    if (cache.find(page) == cache.end()) {
      if (order.size() >= capacity) {
        int farthestPage = -1;
        int farthestDistance = -1;

        for (int cachedPage : cache) {
          auto it = find(futureAccesses.begin(), futureAccesses.end(), cachedPage);
          if (it == futureAccesses.end()) {
            farthestPage = cachedPage;
            break;
          }
          int distance = std::distance(futureAccesses.begin(), it);
          if (distance > farthestDistance) {
            farthestDistance = distance;
            farthestPage = cachedPage;
          }
        }
        order.remove(farthestPage);
        cache.erase(farthestPage);
      }
      order.push_back(page);
      cache.insert(page);
      auto it = find(futureAccesses.begin(), futureAccesses.end(), page);
      if (it != futureAccesses.end()) {
```

```
            futureAccesses.erase(it);
          }

          // Increment the page fault count
          pageFaultCount++;
        }
    }

    void displayCache() {
        cout << "Current Cache: ";
        for (int cachedPage : order) {
            cout << cachedPage << " ";
        }
        cout << "\tPage Faults: " << pageFaultCount << endl;
    }
};

int main() {
    int capacity;
    cout << "Enter the capacity of the cache: ";
    cin >> capacity;

    vector<int> futureAccesses = {1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5};
    OptimalCache optimalCache(capacity, futureAccesses);

    int pages[] = {1, 7,1, 6,7, 4, 5};
    for (int page : pages) {
        optimalCache.accessPage(page);
        optimalCache.displayCache();
    }

    return 0;
}
```

Output:

```
Enter the capacity of the cache: 3
Current Cache: 1        Page Faults: 1
Current Cache: 1 7      Page Faults: 2
Current Cache: 1 7      Page Faults: 2
Current Cache: 1 7 6    Page Faults: 3
Current Cache: 1 7 6    Page Faults: 3
Current Cache: 1 7 4    Page Faults: 4
Current Cache: 1 4 5    Page Faults: 5
```

Experiment- 6A

Aim: Write a program to implement first fit algorithm for memory management.

Theory:

Source Code:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
struct MemoryBlock {
    int start;
    int size;
    bool free;
    int process_id;
};
class MemoryManager {
private:
    int total_memory;
    int available_memory;
    vector<MemoryBlock> memory_blocks;

public:
    MemoryManager(int                total_memory)                :                total_memory(total_memory),
available_memory(total_memory) {
        MemoryBlock initial_block = {0, total_memory, true, -1};
        memory_blocks.push_back(initial_block);
    }
    bool allocate_memory(int process_id, int size) {
        for (auto& block : memory_blocks) {
            if (block.free && block.size >= size) {
                block.free = false;
                block.process_id = process_id;
                available_memory -= size;
                cout << "Allocated " << size << " units of memory to Process " << process_id
                     << " starting at address " << block.start << endl;

                // If the block size is greater than the required size, split the block
                if (block.size > size) {
                    MemoryBlock new_block = {block.start + size, block.size - size, true, -1};
                    auto it = find_if(memory_blocks.begin(), memory_blocks.end(),
                                 [=](const MemoryBlock& b) { return b.start == block.start; });
                    memory_blocks.insert(it + 1, new_block);
                    cout << "Split block. Remaining free memory: " << new_block.size << " units" << endl;
                }

                return true;
            }
        }
        cout << "Unable to allocate " << size << " units of memory for Process " << process_id << endl;
```

```
        return false;
    }
    bool deallocate_memory(int process_id) {
        for (auto& block : memory_blocks) {
            if (!block.free && block.process_id == process_id) {
                block.free = true;
                block.process_id = -1;
                available_memory += block.size;
                cout << "Deallocated memory for Process " << process_id << ". Free memory: " << block.size <<
" units" << endl;
                return true;
            }
        }
        cout << "No memory allocated for Process " << process_id << endl;
        return false;
    }
    void display_memory_status() {
        cout << "\nMemory Status:" << endl;
        cout << "Total Memory: " << total_memory << " units" << endl;
        cout << "Available Memory: " << available_memory << " units" << endl;
        cout << "Memory Blocks:" << endl;
        for (const auto& block : memory_blocks) {
            cout << "Start: " << block.start << ", Size: " << block.size << ", Free: " << block.free
                 << ", Process ID: " << block.process_id << endl;
        }
        cout << "\n";
    }
};
int main() {
    MemoryManager memory_manager(100);
    memory_manager.display_memory_status();
    memory_manager.allocate_memory(1, 20);
    memory_manager.allocate_memory(2, 30);
    memory_manager.deallocate_memory(1);
    memory_manager.display_memory_status();
    return 0;
}
```

Output:

```
Memory Status:
Total Memory: 100 units
Available Memory: 100 units
Memory Blocks:
Start: 0, Size: 100, Free: 1, Process ID: -1

Allocated 20 units of memory to Process 1 starting at address 0
Split block. Remaining free memory: 80 units
Allocated 30 units of memory to Process 2 starting at address 20
Split block. Remaining free memory: 50 units
Deallocated memory for Process 1. Free memory: 100 units

Memory Status:
Total Memory: 100 units
Available Memory: 150 units
Memory Blocks:
Start: 0, Size: 100, Free: 1, Process ID: -1
Start: 20, Size: 80, Free: 0, Process ID: 2
Start: 50, Size: 50, Free: 1, Process ID: -1
```

Experiment- 6B

Aim: Write a program to implement best fit algorithm for memory management.

Theory:

Source Code:

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

struct MemoryBlock {
    int start;
    int size;
    bool free;
    int process_id;

    // Custom equality operator for std::find
    bool operator==(const MemoryBlock& other) const {
        return start == other.start && size == other.size && free == other.free && process_id ==
other.process_id;
    }
};

class MemoryManager {
private:
    int total_memory;
    int available_memory;
    vector<MemoryBlock> memory_blocks;

public:
    MemoryManager(int total_memory) : total_memory(total_memory), available_memory(total_memory)
{
        MemoryBlock initial_block = {0, total_memory, true, -1};
        memory_blocks.push_back(initial_block);
    }

    bool allocate_memory(int process_id, int size) {
        vector<MemoryBlock*> free_blocks;

        for (auto& block : memory_blocks) {
            if (block.free && block.size >= size) {
                free_blocks.push_back(&block);
            }
        }

        if (free_blocks.empty()) {
            cout << "Unable to allocate " << size << " units of memory for Process " << process_id << endl;
            return false;
        }

        // Find the best fit block
        auto best_fit_block = min_element(free_blocks.begin(), free_blocks.end(),
                                  [](const MemoryBlock* a, const MemoryBlock* b) {
```

```
                                    return a->size < b->size;
                                  });

        (*best_fit_block)->free = false;
        (*best_fit_block)->process_id = process_id;
        available_memory -= size;
        cout << "Allocated " << size << " units of memory to Process " << process_id
             << " starting at address " << (*best_fit_block)->start << " (Best Fit)" << endl;

        // If the block size is greater than the required size, split the block
        if ((*best_fit_block)->size > size) {
            MemoryBlock new_block = {(*best_fit_block)->start + size, (*best_fit_block)->size - size, true, -
1};
            auto it = find(memory_blocks.begin(), memory_blocks.end(), *(*best_fit_block));
            memory_blocks.insert(it + 1, new_block);
            cout << "Split block. Remaining free memory: " << new_block.size << " units" << endl;
        }

        return true;
    }

    bool deallocate_memory(int process_id) {
        for (auto& block : memory_blocks) {
            if (!block.free && block.process_id == process_id) {
                block.free = true;
                block.process_id = -1;
                available_memory += block.size;
                cout << "Deallocated memory for Process " << process_id << ". Free memory: " << block.size
<< " units" << endl;
                return true;
            }
        }

        cout << "No memory allocated for Process " << process_id << endl;
        return false;
    }

    void display_memory_status() {
        cout << "\nMemory Status:" << endl;
        cout << "Total Memory: " << total_memory << " units" << endl;
        cout << "Available Memory: " << available_memory << " units" << endl;
        cout << "Memory Blocks:" << endl;
        for (const auto& block : memory_blocks) {
            cout << "Start: " << block.start << ", Size: " << block.size << ", Free: " << block.free
                 << ", Process ID: " << block.process_id << endl;
        }
        cout << "\n";
    }
};

int main() {
```

```
    MemoryManager memory_manager(100);

    memory_manager.display_memory_status();

    memory_manager.allocate_memory(1, 20);
    memory_manager.allocate_memory(2, 30);
    memory_manager.allocate_memory(3, 10);
    memory_manager.deallocate_memory(1);

    memory_manager.display_memory_status();

    return 0;
}
```

Output:

```
Memory Status:
Total Memory: 100 units
Available Memory: 100 units
Memory Blocks:
Start: 0, Size: 100, Free: 1, Process ID: -1

Allocated 20 units of memory to Process 1 starting at address 0 (Best Fit)
Split block. Remaining free memory: 80 units
Allocated 30 units of memory to Process 2 starting at address 20 (Best Fit)
Split block. Remaining free memory: 50 units
Allocated 10 units of memory to Process 3 starting at address 50 (Best Fit)
Split block. Remaining free memory: 40 units
Deallocated memory for Process 1. Free memory: 100 units

Memory Status:
Total Memory: 100 units
Available Memory: 140 units
Memory Blocks:
Start: 0, Size: 100, Free: 1, Process ID: -1
Start: 20, Size: 80, Free: 0, Process ID: 2
Start: 50, Size: 50, Free: 0, Process ID: 3
Start: 60, Size: 40, Free: 1, Process ID: -1
```

Experiment- 6 C

Aim: Write a program to implement worst fit algorithm for memory management.

Theory:

Source Code:

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

struct MemoryBlock {
    int start;
    int size;
    bool free;
    int process_id;

    // Custom comparison operator for std::max_element
    bool operator<(const MemoryBlock& other) const {
        return size < other.size;
    }

    // Custom equality operator for std::find
    bool operator==(const MemoryBlock& other) const {
        return start == other.start && size == other.size && free == other.free && process_id ==
other.process_id;
    }
};

class MemoryManager {
private:
    int total_memory;
    int available_memory;
    vector<MemoryBlock> memory_blocks;

public:
    MemoryManager(int total_memory) : total_memory(total_memory), available_memory(total_memory)
{
        MemoryBlock initial_block = {0, total_memory, true, -1};
        memory_blocks.push_back(initial_block);
    }

    bool allocate_memory(int process_id, int size) {
        vector<MemoryBlock*> free_blocks;

        for (auto& block : memory_blocks) {
            if (block.free && block.size >= size) {
                free_blocks.push_back(&block);
            }
        }

        if (free_blocks.empty()) {
            cout << "Unable to allocate " << size << " units of memory for Process " << process_id << endl;
            return false;
```

```
        }

        // Find the worst fit block
        auto worst_fit_block = max_element(free_blocks.begin(), free_blocks.end(),
                                   [](const MemoryBlock* a, const MemoryBlock* b) {
                                       return *a < *b;
                                   });

        (*worst_fit_block)->free = false;
        (*worst_fit_block)->process_id = process_id;
        available_memory -= size;
        cout << "Allocated " << size << " units of memory to Process " << process_id
             << " starting at address " << (*worst_fit_block)->start << " (Worst Fit)" << endl;

        // If the block size is greater than the required size, split the block
        if ((*worst_fit_block)->size > size) {
            MemoryBlock new_block = {(*worst_fit_block)->start + size, (*worst_fit_block)->size - size,
true, -1};
            auto it = find(memory_blocks.begin(), memory_blocks.end(), *(*worst_fit_block));
            memory_blocks.insert(it + 1, new_block);
            cout << "Split block. Remaining free memory: " << new_block.size << " units" << endl;
        }

        return true;
    }

    bool deallocate_memory(int process_id) {
        for (auto& block : memory_blocks) {
            if (!block.free && block.process_id == process_id) {
                block.free = true;
                block.process_id = -1;
                available_memory += block.size;
                cout << "Deallocated memory for Process " << process_id << ". Free memory: " << block.size
<< " units" << endl;
                return true;
            }
        }
        cout << "No memory allocated for Process " << process_id << endl;
        return false;
    }
    void display_memory_status() {
        cout << "\nMemory Status:" << endl;
        cout << "Total Memory: " << total_memory << " units" << endl;
        cout << "Available Memory: " << available_memory << " units" << endl;
        cout << "Memory Blocks:" << endl;
        for (const auto& block : memory_blocks) {
            cout << "Start: " << block.start << ", Size: " << block.size << ", Free: " << block.free
                 << ", Process ID: " << block.process_id << endl;
        }
        cout << "\n";
    }
```

```
};

int main() {
   MemoryManager memory_manager(100);
   memory_manager.display_memory_status();
   memory_manager.allocate_memory(1, 20);
   memory_manager.allocate_memory(2, 30);
   memory_manager.allocate_memory(3, 10);
   memory_manager.deallocate_memory(1);
   memory_manager.display_memory_status();
   return 0;
}
```

Output:

```
Memory Status:
Total Memory: 100 units
Available Memory: 100 units
Memory Blocks:
Start: 0, Size: 100, Free: 1, Process ID: -1

Allocated 20 units of memory to Process 1 starting at address 0 (Worst Fit)
Split block. Remaining free memory: 80 units
Allocated 30 units of memory to Process 2 starting at address 20 (Worst Fit)
Split block. Remaining free memory: 50 units
Allocated 10 units of memory to Process 3 starting at address 50 (Worst Fit)
Split block. Remaining free memory: 40 units
Deallocated memory for Process 1. Free memory: 100 units

Memory Status:
Total Memory: 100 units
Available Memory: 140 units
Memory Blocks:
Start: 0, Size: 100, Free: 1, Process ID: -1
Start: 20, Size: 80, Free: 0, Process ID: 2
Start: 50, Size: 50, Free: 0, Process ID: 3
Start: 60, Size: 40, Free: 1, Process ID: -1
```

Experiment- 7

Aim: Write a program to implement reader/writer problem using semaphore.

Theory:

Source Code:

```
#include <iostream>
#include <thread>
#include <semaphore>

using namespace std;

const int num_readers = 5;
const int num_writers = 2;

string shared_data = "";
int readers_count = 0;

// Semaphores
std::binary_semaphore write_mutex(1);
std::binary_semaphore read_mutex(1);
std::binary_semaphore readers_count_mutex(1);

void writer() {
    // Acquire the write mutex to ensure exclusive access to shared_data
    write_mutex.acquire();

    cout << "Writer is writing data." << endl;
    shared_data = "New data";

    // Release the write mutex to allow other writers or readers
    write_mutex.release();
}

void reader(int reader_id) {
    // Acquire the mutex to ensure mutual exclusion for updating readers_count
    readers_count_mutex.acquire();
    readers_count++;

    // If this is the first reader, acquire the read mutex to block writers
    if (readers_count == 1) {
        read_mutex.acquire();
    }

    // Release the mutex for readers_count
    readers_count_mutex.release();

    // Read data
    cout << "Reader " << reader_id << " is reading data: " << shared_data << endl;

    // Acquire the mutex to ensure mutual exclusion for updating readers_count
    readers_count_mutex.acquire();
    readers_count--;

    // If this is the last reader, release the read mutex to allow writers
    if (readers_count == 0) {
```

```
        read_mutex.release();
    }

    // Release the mutex for readers_count
    readers_count_mutex.release();
}

int main() {
    // Create reader threads
    thread reader_threads[num_readers];
    for (int i = 0; i < num_readers; ++i) {
        reader_threads[i] = thread(reader, i);
    }
    // Create writer threads
    thread writer_threads[num_writers];
    for (int i = 0; i < num_writers; ++i) {
        writer_threads[i] = thread(writer);
    }
    // Wait for all threads to finish
    for (int i = 0; i < num_readers; ++i) {
        reader_threads[i].join();
    }
    for (int i = 0; i < num_writers; ++i) {
        writer_threads[i].join();
    }
    cout << "All threads finished." << endl;
    return 0;
}
```

Output:

```
Reader Reader 2Reader 4 is reading data:  is reading data:
Reader 3 is reading data:
0 is reading data:

Writer is writing data.
Writer is writing data.
Reader 1 is reading data: New data
All threads finished.
```

Experiment- 8

Aim: Write a program to implement Producer-Consumer problem using semaphores.
Theory:

Source Code:

```
#define _WIN32_WINNT 0x0601
#include <iostream>
#include <thread>
#include <vector>
#include <semaphore>
#include <queue>
using namespace std;
const int buffer_size = 5;
const int num_producers = 2;
const int num_consumers = 2;
queue<int> buffer;
std::binary_semaphore empty_slots(buffer_size);
std::binary_semaphore full_slots(0);
std::binary_semaphore buffer_mutex(1);
void producer(int producer_id) {
    for (int i = 0; i < 10; ++i) {
        int item = rand() % 100;  // Generate a random item
        empty_slots.acquire();    // Wait for an empty slot
        buffer_mutex.acquire();   // Acquire buffer access
        buffer.push(item);
        cout << "Producer " << producer_id << " produced item: " << item << " (Buffer size: " <<
buffer.size() << ")" << endl;
        buffer_mutex.release();   // Release buffer access
        full_slots.release();     // Signal that a slot is full
        // Simulate some work being done before producing the next item
        this_thread::sleep_for(chrono::milliseconds(rand() % 100));
    }
}

void consumer(int consumer_id) {
    for (int i = 0; i < 10; ++i) {
        full_slots.acquire();     // Wait for a full slot
        buffer_mutex.acquire();   // Acquire buffer access

        int item = buffer.front();
        buffer.pop();
        cout << "Consumer " << consumer_id << " consumed item: " << item << " (Buffer size: " <<
buffer.size() << ")" << endl;

        buffer_mutex.release();   // Release buffer access
        empty_slots.release();    // Signal that a slot is empty

        // Simulate some work being done before consuming the next item
        this_thread::sleep_for(chrono::milliseconds(rand() % 100));
    }
}

int main() {
    // Create producer threads
    vector<thread> producer_threads;
```

```
    for (int i = 0; i < num_producers; ++i) {
        producer_threads.emplace_back(producer, i);
    }
    // Create consumer threads
    vector<thread> consumer_threads;
    for (int i = 0; i < num_consumers; ++i) {
        consumer_threads.emplace_back(consumer, i);
    }
    // Wait for all producer threads to finish
    for (auto& thread : producer_threads) {
        thread.join();
    }
    // Wait for all consumer threads to finish
    for (auto& thread : consumer_threads) {
        thread.join();
    }
    cout << "All threads finished." << endl;

    return 0;
}
```

Output:

```
Producer 1 produced item: 41 (Buffer size: 1)
Producer 0 produced item: 41 (Buffer size: 2)
Consumer 1 consumed item: 41 (Buffer size: 1)
Consumer 0 consumed item: 41 (Buffer size: 0)
Producer 0 produced item: 34 (Buffer size: 1)
Producer 1 produced item: 34 (Buffer size: 2)
Producer 0 produced item: 69 (Buffer size: 3)
Consumer 0 consumed item: 34 (Buffer size: 2)
Producer 1 produced item: 69 (Buffer size: 3)
Consumer 1 consumed item: 34 (Buffer size: 2)
Producer 1 produced item: 78 (Buffer size: 3)
Producer 0 produced item: 78 (Buffer size: 4)
Consumer 0 consumed item: 69 (Buffer size: 3)
Consumer 1 consumed item: 69 (Buffer size: 2)
Producer 0 produced item: 62 (Buffer size: 3)
Producer 1 produced item: 62 (Buffer size: 4)
Consumer 1 consumed item: 78 (Buffer size: 3)
Consumer 0 consumed item: 78 (Buffer size: 2)
Consumer 0 consumed item: 62 (Buffer size: 1)
Consumer 1 consumed item: 62 (Buffer size: 0)
Producer 1 produced item: 5 (Buffer size: 1)
Producer 0 produced item: 5 (Buffer size: 2)
Consumer 0 consumed item: 5 (Buffer size: 1)
Consumer 1 consumed item: 5 (Buffer size: 0)
Producer 1 produced item: 81 (Buffer size: 1)
Consumer 1 consumed item: 81 (Buffer size: 0)
Producer 0 produced item: 81 (Buffer size: 1)
Consumer 0 consumed item: 81 (Buffer size: 0)
Producer 1 produced item: 61 (Buffer size: 1)
```

```
Producer 0 produced item: 61 (Buffer size: 2)
Consumer 0 consumed item: 61 (Buffer size: 1)
Consumer 1 consumed item: 61 (Buffer size: 0)
Producer 1 produced item: 95 (Buffer size: 1)
Producer 0 produced item: 95 (Buffer size: 2)
Consumer 0 consumed item: 95 (Buffer size: 1)
Consumer 1 consumed item: 95 (Buffer size: 0)
Producer 1 produced item: 27 (Buffer size: 1)
Producer 0 produced item: 27 (Buffer size: 2)
Consumer 0 consumed item: 27 (Buffer size: 1)
Consumer 1 consumed item: 27 (Buffer size: 0)
All threads finished.
```

Experiment- 9

Aim: Write a program to implement Banker’s algorithm for deadlock avoidance.

Theory:

Source Code:

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

class BankersAlgorithm {
private:
    int processes;
    int resources;
    vector<vector<int>> max_claim;
    vector<vector<int>> allocation;
    vector<vector<int>> need;
    vector<int> available;
    vector<int> work;
    vector<bool> finish;
    vector<int> sequence;

    bool isSafeState(int process) {
        for (int i = 0; i < resources; ++i) {
            if (need[process][i] > work[i]) {
                return false;
            }
        }
        return true;
    }

    void execute() {
        while (true) {
            bool found = false;
            for (int i = 0; i < processes; ++i) {
                if (!finish[i] && isSafeState(i)) {
                    for (int j = 0; j < resources; ++j) {
                        work[j] += allocation[i][j];
                    }
                    finish[i] = true;
                    sequence.push_back(i);
                    found = true;

                    // Print safe state
                    cout << "Safe state after process " << i << ": ";
                    for (int j = 0; j < resources; ++j) {
                        cout << work[j] << " ";
                    }
                    cout << endl;
                }
            }
            if (!found) {
                break;
```

```
        }
    }
}

public:
    BankersAlgorithm(int p, int r, vector<vector<int>> &max_claim, vector<vector<int>> &allocation)
        : processes(p), resources(r), max_claim(max_claim), allocation(allocation), finish(processes, false) {

        // Initialize need matrix
        need.resize(processes, vector<int>(resources, 0));
        for (int i = 0; i < processes; ++i) {
            for (int j = 0; j < resources; ++j) {
                need[i][j] = max_claim[i][j] - allocation[i][j];
            }
        }

        // Initialize available and work vectors
        available.resize(resources, 0);
        work.resize(resources, 0);
        for (int j = 0; j < resources; ++j) {
            for (int i = 0; i < processes; ++i) {
                available[j] += allocation[i][j];
            }
            work[j] = available[j];
        }
    }

    void runAlgorithm() {
        // Print initial available resources
        cout << "Initial available resources: ";
        for (int i : work) {
            cout << i << " ";
        }
        cout << endl;

        execute();

        if (all_of(finish.begin(), finish.end(), [](bool f) { return f; })) {
            cout << "Safe state" << endl;
            cout << "Safe sequence: ";
            for (int i : sequence) {
                cout << i << " ";
            }
            cout << endl;
        } else {
            cout << "Unsafe state" << endl;
        }
    }
};

// Example usage
```

```
int main() {
    int processes = 5;
    int resources = 3;
    vector<vector<int>> max_claim = {
        {7, 5, 3},
        {3, 2, 2},
        {9, 0, 2},
        {2, 2, 2},
        {4, 3, 3}
    };
    vector<vector<int>> allocation = {
        {0, 1, 0},
        {2, 0, 0},
        {3, 0, 2},
        {2, 1, 1},
        {0, 0, 2}
    };

    BankersAlgorithm banker(processes, resources, max_claim, allocation);
    banker.runAlgorithm();

    return 0;
}
```

Output:

```
Initial available resources: 7 2 5
Safe state after process 1: 9 2 5
Safe state after process 2: 12 2 7
Safe state after process 3: 14 3 8
Safe state after process 4: 14 3 10
Unsafe state
```