1. Process Creation and termination for operating system(fork , wait , signal , etc.).

System calls are the interface between user-level applications and the operating system (OS) kernel. They are the mechanisms that allow applications to request services from the operating system, such as file I/O, network communication, process management, and memory allocation.

When an application makes a system call, it executes a special instruction that switches the CPU from user mode to kernel mode. In kernel mode, the CPU has unrestricted access to the hardware and can execute privileged instructions that are not available in user mode. The kernel then performs the requested operation on behalf of the application and returns control to the application in user mode.

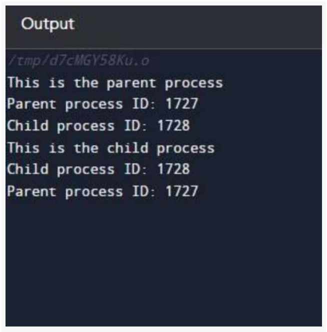
Some common system calls include:

open(): Opens a file or creates a new file. read(): Reads data from a file. write(): Writes data to a file. close(): Closes a file. fork(): Creates a new process by duplicating the current process. exec(): Replaces the current process with a new process. wait(): Waits for a child process to terminate.
exit(): Terminates the current process. socket(): Creates a new network socket. connect(): Establishes a connection with a remote host.
System calls are essential for building and running applications on an operating system, as they provide a standardized way for applications to interact with the underlying hardware and services.

Code for process Creation and Termination:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
  pid_t pid;
  int status;
  pid = fork(); // create a child process

  if (pid < 0) {
    // fork failed
     fprintf(stderr, "Fork failed\n");
     return 1;
  } else if (pid == 0) {
    // child process
     printf("This is the child process\n");
     printf("Child process ID: %d\n", getpid());
    printf("Parent process ID: %d\n", getppid());
     // do some work here
     sleep(5);
    printf("Child process completed\n");
    exit(0);
  } else {
     // parent process
     printf("This is the parent process\n");
    printf("Parent process ID: %d\n", getpid());
     printf("Child process ID: %d\n", pid);
    // wait for the child process to complete
     wait(&status);
    printf("Child process terminated with status %d\n", status);
  }

  return 0;
}
```

Output->



Output

```
/tmp/d7cMGY58Ku.o
This is the parent process
Parent process ID: 1727
Child process ID: 1728
This is the child process
Child process ID: 1728
Parent process ID: 1727
```

2. Implementing CPU SCheduling algorithms : FCFS , SJF , Round Robin ,preemptive priority .

CPU scheduling algorithms are methods used by the operating system to determine which process should run on the CPU next. There are several different scheduling algorithms that can be used, including:

1.) First-Come, First-Serve (FCFS) - In this algorithm, the process that arrives first is served first. The operating system simply schedules processes in the order they arrive.

2.) Shortest Job First (SJF) - This algorithm schedules processes based on their expected processing time. The process with the shortest expected processing time is scheduled first, in order to minimize the average waiting time for all processes.

3.) Round Robin (RR) - This algorithm assigns a fixed time slice to each process in turn, allowing each process to run for a predetermined amount of time before being preempted and moved to the back of the queue.

4.) Preemptive Priority - This algorithm assigns a priority value to each process, and the process with the highest priority is scheduled to run first. In the preemptive version of this algorithm, the operating system can preempt a lower-priority process if a higher-priority process becomes available.

Implementing these scheduling algorithms involves writing code to determine which process should be scheduled next based on the algorithm being used.
Here are some basic steps for each algorithm:

1.) FCFS:

Maintain a queue of processes in the order they arrive.
When the CPU becomes available, schedule the next process in the queue.

2.) SJF:

Maintain a queue of processes sorted by expected processing time. When the CPU becomes available, schedule the process with the shortest expected processing time.

3.) RR:

Maintain a queue of processes.

Assign a fixed time slice to each process.

When the CPU becomes available, schedule the next process in the queue, and run it for the fixed time slice.

Move the process to the back of the queue.

4.) Preemptive Priority:

Maintain a queue of processes sorted by priority.

When the CPU becomes available, schedule the highest-priority process. If a higher-priority process becomes available, preempt the currently running process and schedule the higher-priority process.

Code for FCFS:

```
void fcfs(std::vector<Process> processes) {
    std::queue<Process> queue;
    for (Process process : processes) {
        queue.push(process);
    }

    while (!queue.empty()) {
        Process current_process = queue.front();
        queue.pop();
        current_process.execute();
    }
}
```

Code for SJF:

```
void sjf(std::vector<Process> processes) {
    std::vector<Process> queue;
    for (Process process : processes) {
        queue.push_back(process);
    }

    std::sort(queue.begin(), queue.end(), [](Process a, Process b) {
        return a.expected_time < b.expected_time;
    });

    while (!queue.empty()) {
        Process current_process = queue.front();
        queue.erase(queue.begin());
        current_process.execute();
    }
}
```

Code for Round Robin:

```
void rr(std::vector<Process> processes, int time_slice) {
    std::queue<Process> queue;
    for (Process process : processes) {
        queue.push(process);
    }

    while (!queue.empty()) {
        Process current_process = queue.front();
        queue.pop();
        if (current_process.expected_time <= time_slice) {
            current_process.execute();
        } else {
            current_process.execute_for(time_slice);
            queue.push(current_process);
        }
    }
}
```

Code for Preemptive Priority:

```
void priority(std::vector<Process> processes) {
    std::vector<Process> queue;
    for (Process process : processes) {
        queue.push_back(process);
    }

    std::sort(queue.begin(), queue.end(), [](Process a, Process b) {
        return a.priority > b.priority;
    });

    while (!queue.empty()) {
        Process current_process = queue.front();
        queue.erase(queue.begin());
        current_process.execute();

        if (!queue.empty() && queue.front().priority > current_process.priority) {
            queue.push_back(current_process);
            std::sort(queue.begin(), queue.end(), [](Process a, Process b) {
                return a.priority > b.priority;
            });
        }
    }
}
```

3. Inter Process Communication.

Inter-process communication (IPC) refers to the methods and mechanisms used by different processes running on a computer system to communicate and exchange data with each other.

Shared memory and message passing are two common ways to implement IPC:

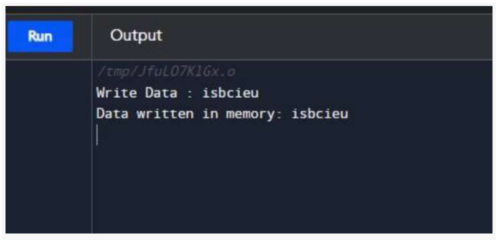
1.) Shared Memory:

In shared memory, a region of memory is allocated that can be accessed by multiple processes. Each process can read and write data to this shared memory space. The data can be shared between processes quickly and efficiently as there is no need to copy data between processes.

Implementation in cpp:

```
#include <iostream>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>  using
namespace std;

int main()
{
     // ftok to generate unique key
     key_t key = ftok("shmfile",65);
    // shmget returns an identifier in shmid
    int shmid = shmget(key,1024,0666|IPC_CREAT);
    // shmat to attach to shared memory
    char *str = (char*) shmat(shmid,(void*)0,0);

    cout<<"Write Data : ";
    gets(str);

    printf("Data written in memory: %s\n",str);

    //detach from shared memory
    shmdt(str);
    return 0;
}
```

Output->



```
/tmp/JfuLO7K1Gx.o
Write Data : isbcieu
Data written in memory: isbcieu
```

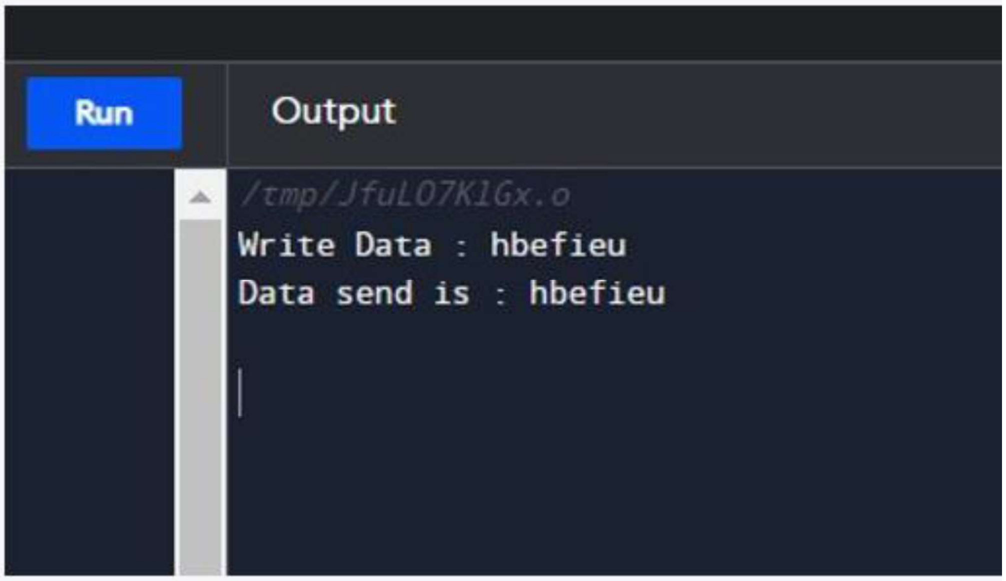
2.) Message Passing

Message passing involves sending messages between processes using a common communication channel. The processes can send and receive messages through this channel, and the communication can be either synchronous or asynchronous.

Implementation in cpp:

```
 // C Program for Message Queue (Writer Process)
 #include <stdio.h>
 #include <sys/ipc.h>
 #include <sys/msg.h>
#define MAX 10

 // structure for message queue  struct
mesg_buffer {
    long mesg_type;
    char mesg_text[100];
 } message;

 int main()
 {
    key_t key;
    int msgid;

    // ftok to generate unique key
    key = ftok("progfile", 65);
   // msgget creates a message queue
   // and returns identifier
   msgid = msgget(key, 0666 | IPC_CREAT);
    message.mesg_type = 1;

   printf("Write Data : ");
    fgets(message.mesg_text,MAX,stdin);

   // msgsnd to send message
   msgsnd(msgid, &message, sizeof(message), 0);

   // display the message
   printf("Data send is : %s \n", message.mesg_text);

   return 0;
   }
```

Output->

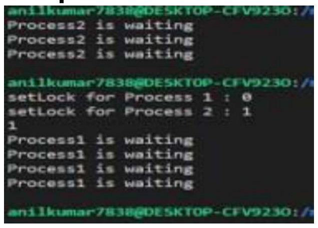


```
/tmp/JfuLO7K1Gx.o
Write Data : hbefieu
Data send is : hbefieu
```

4. Critical Section

```
 #include <stdio.h>
#include <pthread.h> int
shared_resource = 0;
          pthread_mutex_t
mutex;

void *increment(void *arg) {           for (int i =
          0; i < 1000000; i++) {
          pthread_mutex_lock(&mutex);
          Shared_resource++;
          pthread_mutex_unlock(&mutex);
            }
           return NULL;
}
 int main() {
           pthread_mutex_init(&mutex, NULL);  pthread_t t1,

          t2;  pthread_create(&t1, NULL, increment, NULL);

          pthread_create(&t2, NULL, increment, NULL);

          pthread_join(t1, NULL);  pthread_join(t2, NULL);

          printf("Shared resource = %d\n", shared_resource);

          pthread_mutex_destroy(&mutex);

           return 0;

}
```

Output->



```
anilkumar7838@DESKTOP-CFV9230:/
Process2 is waiting
Process2 is waiting
Process2 is waiting

anilkumar7838@DESKTOP-CFV9230:/
setLock for Process 1 : 0
setLock for Process 2 : 1
1
Process1 is waiting
Process1 is waiting
Process1 is waiting
Process1 is waiting

anilkumar7838@DESKTOP-CFV9230:/
```

When more than one process try to access the same code segment that segment is known as the critical section. The critical section contains shared variables or resources which are needed to be synchronized to maintain the consistency of data variables.

In simple terms, a critical section is a group of instructions/statements or regions of code that need to be executed atomically , such as accessing a resource (file, input or output port, global data, etc.) In concurrent programming, if one thread tries to change the value of shared data at the same time as another thread tries to read the value (i.e, data race across threads), the result is unpredictable. The access to such shared variables (shared memory, shared files, shared port, etc.) is to be synchronized.

5. Reader- Writers problem and Dining Philosphers problem using Semaphors.

Code: Reader Writers

```
 #include <stdlib.h>

 #include <stdio.h>

 #include <pthread.h>  #include <semaphore.h>

sem_t w;      // write access  sem_t m;          //

mutex  int rc=0;   // readers count  int

writersCount;  int readersCount;  pthread_t

writersThread[100], readersThread[100];  int

writeCount[10], readCount[10]  void *writer(void *i)

{

          int a = *((int *) i);


    sem_wait(&w);   // P(w)  printf("Writer

   %d writes to DB.\n",a+1);

   writeCount[a + 1]++;

    sem_post(&w);   // V(w)

         return NULL;

}

void *reader(void *i)
```

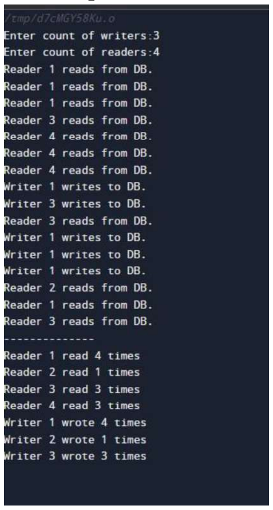
```
{
        int a = *((int *) i);
   sem_wait(&m);   // P(m)
      rc++;  if (rc == 1) {
      sem_wait(&w);   //
      P(w)
        }
   sem_post(&m);   // V (m)


   printf("Reader %d reads from DB.\n", a +
  1);  readCount[a + 1]++;  sem_wait(&m);   //
  P(m)
        rc--;  if (rc == 0) {
      sem_post(&w);   //
      V(w)
        }
   sem_post(&m);   // V(m)
       return NULL;}

int randomCount()
{
       return 5.0 * rand() / RAND_MAX;
}
```

```
int main()

{

    sem_init(&w,0,1);

   sem_init(&m,0,1);

         int i;

         printf("Enter count of writers:");

   scanf("%d",&writersCount);

   printf("Enter count of readers:");

   scanf("%d",&readersCount);  int

   readerIndices[readersCount];  int

   writerIndices[writersCount];  int

   totalReaders = 0;  int totalWriters =

   0;  for (i=0; i<readersCount; i++)

         {

         int j;

         int count;

       readerIndices[i] = i;  count

       = randomCount();


         for (j = 0 ; j < count ; ++j)

         {

          pthread_create(&readersThread[totalReaders++], NULL, reader,
&readerIndices[i]);
```

```
        }

        }

        for (i = 0 ; i < writersCount ; i++)

        {

        int j;

        int count;

      writerIndices[i] = i;

       count          = randomCount();

        for (j = 0 ; j < count ; ++j)

        {

          pthread_create(&writersThread[totalWriters++], NULL, writer,
&writerIndices[i]);

        }

        }

        for (i = 0 ; i < totalWriters ; i++)

        {

       pthread_join(writersThread[i], NULL);

        }

        for (i = 0 ; i < totalReaders ; i++)

        {

       pthread_join(readersThread[i], NULL);

        }
```

```
    printf("--------------\n");  for (i = 0 ; i <
        readersCount ; i++)
         {
       printf("Reader %d read %d times\n", i + 1, readCount[i + 1]);
         }
         for (i = 0 ; i < writersCount ; i++)
         {
       printf("Writer %d wrote %d times\n", i + 1, readCount[i + 1]);
         }
    sem_destroy(&w);
   sem_destroy(&m);
         return 0;
}
```

Output->



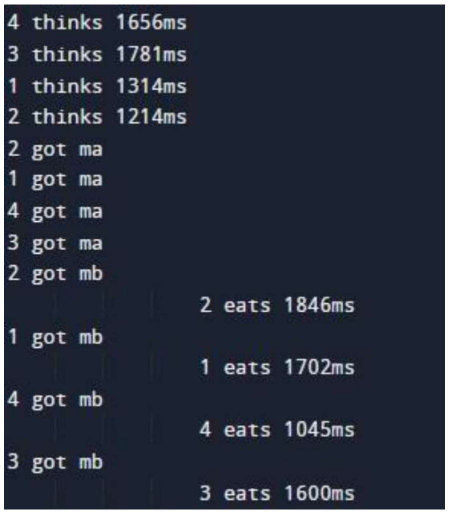
```
/tmp/d7cMGY58Ku.o
Enter count of writers:3
Enter count of readers:4
Reader 1 reads from DB.
Reader 1 reads from DB.
Reader 1 reads from DB.
Reader 3 reads from DB.
Reader 4 reads from DB.
Reader 4 reads from DB.
Reader 4 reads from DB.
Writer 1 writes to DB.
Writer 3 writes to DB.
Reader 3 reads from DB.
Writer 1 writes to DB.
Writer 1 writes to DB.
Writer 1 writes to DB.
Reader 2 reads from DB.
Reader 1 reads from DB.
Reader 3 reads from DB.
--------------
Reader 1 read 4 times
Reader 2 read 1 times
Reader 3 read 3 times
Reader 4 read 3 times
Writer 1 wrote 4 times
Writer 2 wrote 1 times
Writer 3 wrote 3 times
```

Code: Dining Philosophers Problem

```
// dp_1.cpp
#include <iostream>
#include <thread>  #include
<chrono>  int myrand(int min,
int max) {  return rand()%(max-
min)+min;
}


void lock(int& m) {
 m=1;
}


void unlock(int& m) {
  m=0;
}
void phil(int ph, int& ma, int& mb) {  while(true) {  int
 duration=myrand(1000, 2000);  std::cout<<ph<<" thinks
 "<<duration<<"ms\n";
 std::this_thread::sleep_for(std::chrono::milliseconds(duration));
```

```
        lock(ma);  std::cout<<"\t\t"<<ph<<" got ma\n";
  std::this_thread::sleep_for(std::chrono::milliseconds(1000));
  lock(mb);  std::cout<<"\t\t"<<ph<<" got mb\n";
  duration=myrand(1000, 2000);  std::cout<<"\t\t\t\t"<<ph<<"
  eats "<<duration<<"ms\n";
  std::this_thread::sleep_for(std::chrono::milliseconds(duration));
  unlock(mb);  unlock(ma);
 }
}
int main() {
 std::cout<<"dp_1\n";
 srand(time(nullptr));  int
 m1{0}, m2{0}, m3{0}, m4{0};
 std::thread t1([&] {phil(1, m1,
 m2);});  std::thread t2([&]
 {phil(2, m2, m3);});
 std::thread t3([&] {phil(3, m3,
 m4);});  std::thread t4([&]
 {phil(4, m4, m1);});

  t1.join();
 t2.join();
 t3.join();

 t4.join();

}
```

OUTPUT->



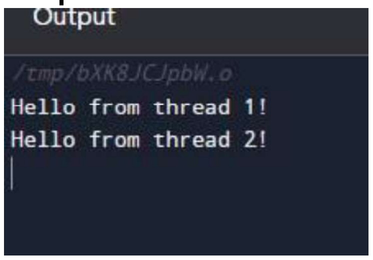
```
4 thinks 1656ms
3 thinks 1781ms
1 thinks 1314ms
2 thinks 1214ms
2 got ma
1 got ma
4 got ma
3 got ma
2 got mb
                2 eats 1846ms
1 got mb
                1 eats 1702ms
4 got mb
                4 eats 1045ms
3 got mb
                3 eats 1600ms
```

6. Thread

A thread is a basic unit of execution in a program, representing a sequence of instructions that can be scheduled for execution independently of the main program flow. Threads can run concurrently, allowing a program to perform multiple tasks simultaneously. In C, threads can be implemented using the pthread library. This library provides functions for creating and managing threads, such as pthread_create(), pthread_join(), and pthread_exit().

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

 void* print_message(void* arg) {
   char* message = (char*) arg;
   printf("%s\n", message);
   pthread_exit(NULL);
 }

 int main() {  pthread_t
   thread1, thread2;

    char* message1 = "Hello from thread 1!";
   pthread_create(&thread1, NULL, print_message, (void*) message1);

    char* message2 = "Hello from thread 2!";
   pthread_create(&thread2, NULL, print_message, (void*) message2);

    pthread_join(thread1, NULL);
   pthread_join(thread2, NULL);

    return 0;
 }
```

Output->



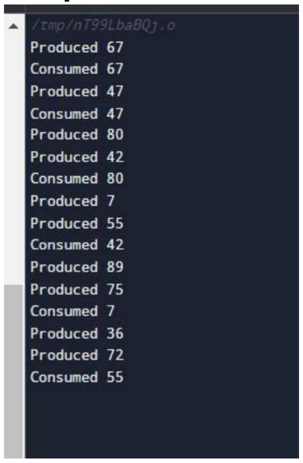
```
/tmp/bXK8JCJpbW.o
Hello from thread 1!
Hello from thread 2!
```

7. Producer- Consumer Problem

Theory: The producer-consumer problem is a classical synchronization problem in computer science. It involves two types of threads, namely producers and consumers, which share a common resource, typically a bounded or unbounded buffer. The producers generate data and put it into the buffer, while the consumers remove the data from the buffer and process it. **Code:**

```
 #include <iostream>
 #include <queue>
 #include <mutex>
 #include <condition_variable>
 #include <thread>  #include

<chrono>  const int

BUFFER_SIZE = 5;

 std::queue<int> buffer;  std::mutex
buffer_mutex;  std::condition_variable
buffer_not_full;  std::condition_variable
buffer_not_empty;

 void producer(int n) {  for
   (int i = 0; i < n; ++i) {  int
   item = rand() % 100;
        {
            std::unique_lock<std::mutex> lock(buffer_mutex);
           buffer_not_full.wait(lock, []{ return buffer.size() < BUFFER_SIZE;
           });  buffer.push(item);  std::cout << "Produced " << item <<
           std::endl;
        }
        buffer_not_empty.notify_one();
       std::this_thread::sleep_for(std::chrono::milliseconds(500));
    }
}
 void consumer(int n) {
   for (int i = 0; i < n; ++i) {
        {
            std::unique_lock<std::mutex> lock(buffer_mutex);
           buffer_not_empty.wait(lock, []{ return !buffer.empty();
           }); int item = buffer.front(); buffer.pop(); std::cout <<
           "Consumed " << item << std::endl;
        }
        buffer_not_full.notify_one();
       std::this_thread::sleep_for(std::chrono::milliseconds(1000));
    }
}

int main() {
    srand(time(NULL));

    std::thread producer_thread(producer, 10);
   std::thread consumer_thread(consumer, 10);

    producer_thread.join();
   consumer_thread.join();

    return 0;
}
```

Output->

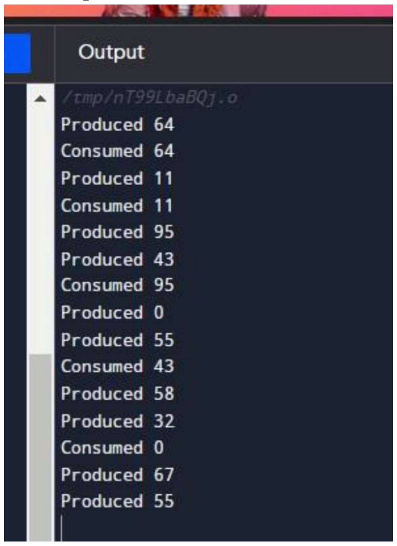


```
/tmp/nT99LbaBQj.o
Produced 67
Consumed 67
Produced 47
Consumed 47
Produced 80
Produced 42
Consumed 80
Produced 7
Produced 55
Consumed 42
Produced 89
Produced 75
Consumed 7
Produced 36
Produced 72
Consumed 55
```

```
#include <iostream>
#include <queue>
#include <mutex>
#include <condition_variable>
```

```
 #include <thread>
#include <chrono>

 std::queue<int> buffer;  std::mutex
buffer_mutex;  std::condition_variable
buffer_not_empty;

 void producer(int n) {  for
    (int i = 0; i < n; ++i) {
          int item = rand() % 100;
          {
              std::unique_lock<std::mutex>
             lock(buffer_mutex);  buffer.push(item);  std::cout
             << "Produced " << item << std::endl;
          }
          buffer_not_empty.notify_one();
         std::this_thread::sleep_for(std::chrono::milliseconds(500));
     }
 }

 void consumer() {
     while (true) {
          {
              std::unique_lock<std::mutex> lock(buffer_mutex);
             buffer_not_empty.wait(lock, []{ return !buffer.empty();
             });  int item = buffer.front();  buffer.pop();  std::cout <<
             "Consumed " << item << std::endl;
          }
          std::this_thread::sleep_for(std::chrono::milliseconds(1000));
     }
 }

 int main() {
     srand(time(NULL));

     std::thread producer_thread(producer, 10);
    std::thread consumer_thread(consumer);
    producer_thread.join();  consumer_thread.join();

     return 0;
 }
```

Output->



Output

```
/tmp/nT99LbaBQj.o
Produced 64
Consumed 64
Produced 11
Consumed 11
Produced 95
Produced 43
Consumed 95
Produced 0
Produced 55
Consumed 43
Produced 58
Produced 32
Consumed 0
Produced 67
Produced 55
```

8. Banker’s Algorithm

Theory:

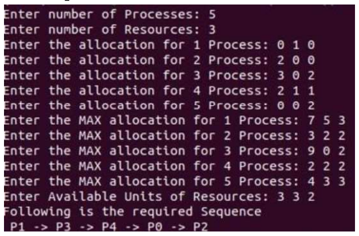
The Banker's algorithm is a deadlock-avoidance algorithm that is used to manage the allocation of resources to processes in a system. It was proposed by Edsger W. Dijkstra in 1965, and it is based on the idea of creating a safe sequence of resource allocations that will not lead to a deadlock.

The algorithm works by maintaining a state of the currently available resources and the current resource allocation to each process. When a process requests a resource, the algorithm checks whether the allocation of the resource will result in a safe state, i.e., a state where no deadlock can occur. If the requested allocation is safe, the algorithm grants the request, updates the resource allocation state, and continues. If the allocation is not safe, the request is denied, and the process must wait until it can be safely granted.

Code:

```
 #include <iostream>
 #include <vector>
 using namespace std;

 const int MAX_PROCESSES = 10;
const int MAX_RESOURCES = 10;

 vector<int> available(MAX_RESOURCES);  vector<vector<int>>
maximum(MAX_PROCESSES, vector<int>(MAX_RESOURCES));
vector<vector<int>> allocation(MAX_PROCESSES, vector<int>(MAX_RESOURCES));
vector<vector<int>> need(MAX_PROCESSES, vector<int>(MAX_RESOURCES));
vector<int> safe_sequence;

 bool is_safe_state() {  vector<bool>
   finished(MAX_PROCESSES, false);  vector<int>
   work = available;

    bool found;  do {  found = false;  for (int i
   = 0; i < maximum.size(); i++) {  if
   (!finished[i] && need[i] <= work) {
   finished[i] = true;  work += allocation[i];
   safe_sequence.push_back(i);  found =
   true;
          }
       }
    } while (found);

    for (int i = 0; i < finished.size(); i++)
      {  if (!finished[i]) {  return false;
       }
    }

    return true;
}

int main() {
    int num_processes, num_resources;
   cout << "Enter the number of processes:
   ";  cin >> num_processes;  cout << "Enter
   the number of resources: ";  cin >>
   num_resources;

    cout << "Enter the available resources:
   ";  for (int i = 0; i < num_resources; i++) {
   cin >> available[i];
    }

    cout << "Enter the maximum resources for each process: " <<
   endl;  for (int i = 0; i < num_processes; i++) {  for (int j = 0; j <
   num_resources; j++) {  cin >> maximum[i][j];  need[i][j] =
   maximum[i][j];
       }
    }

    cout << "Enter the current allocation of resources for each process: " <<
   endl;  for (int i = 0; i < num_processes; i++) {  for (int j = 0; j <
   num_resources; j++) {  cin >> allocation[i][j];  need[i][j] -= allocation[i][j];
       }
    }

    if (is_safe_state()) {  cout << "Safe sequence:
      ";  for (int i = 0; i < safe_sequence.size();
        i++) {  cout << "P" << safe_sequence[i] << "
        ";
         }
         cout << endl;
    } else {
        cout << "No safe sequence exists." << endl;
    }

    return 0;
}
```

Output->



```
Enter number of Processes: 5
Enter number of Resources: 3
Enter the allocation for 1 Process: 0 1 0
Enter the allocation for 2 Process: 2 0 0
Enter the allocation for 3 Process: 3 0 2
Enter the allocation for 4 Process: 2 1 1
Enter the allocation for 5 Process: 0 0 2
Enter the MAX allocation for 1 Process: 7 5 3
Enter the MAX allocation for 2 Process: 3 2 2
Enter the MAX allocation for 3 Process: 9 0 2
Enter the MAX allocation for 4 Process: 2 2 2
Enter the MAX allocation for 5 Process: 4 3 3
Enter Available Units of Resources: 3 3 2
Following is the required Sequence
 P1 -> P3 -> P4 -> P0 -> P2
```

9.Page Replacement Algorithms

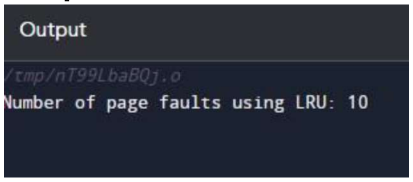
LRU-

The idea behind this algorithm is that the page that has not been used for a long time is less likely to be used again in the near future. The LRU algorithm requires maintaining a queue of all the pages in the main memory, with the most recently used page at the front of the queue and the least recently used page at the back of the queue. When a page fault occurs, the page at the back of the queue is replaced with the new page.

Code:

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

namespace std;

int lru_page_replacement(int pages[], int n, int frame_size)
   {  unordered_map<int, list<int>::iterator> map;  list<int>
   page_list;  int page_faults = 0;

    for (int i = 0; i < n; i++) {  if
       (map.find(pages[i]) != map.end()) {
           // page is already in the memory, move it to the front of the list
          page_list.erase(map[pages[i]]);
       } else {
           // page is not in the memory, remove the least recently used page
          if (page_list.size() == frame_size) {
                int last_page = page_list.back();
               page_list.pop_back();
               map.erase(last_page);
            }
            page_faults++;
        }

        // add the current page to the front of the list
       page_list.push_front(pages[i]);
       map[pages[i]] = page_list.begin();
    }

    return page_faults;
}

int main() {  int pages[] = {1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5};  int n =
   sizeof(pages) / sizeof(pages[0]);  int frame_size = 3;  int page_faults
   = lru_page_replacement(pages, n, frame_size);  cout << "Number of
   page faults using LRU: " << page_faults << endl;

    return 0;
}
```

Output->



```
Output
/tmp/nT99LbaBQj.o
Number of page faults using LRU: 10
```

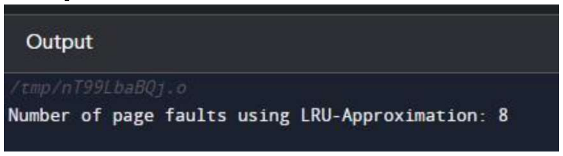
LRU-APPROXIMATION

This algorithm is similar to LRU but does not require maintaining a queue of all the pages in the main memory. Instead, it uses a counter for each page that is incremented every time the page is referenced. When a page fault occurs, the algorithm replaces the page with the lowest counter value.

Code:

```
 #include <iostream>

#include <unordered_map>

using namespace std;

 int lru_approximation_page_replacement(int pages[], int n, int frame_size)

   {  unordered_map<int, int> map;  int page_faults = 0;

    for (int i = 0; i < n; i++) {  if
       (map.find(pages[i]) != map.end()) {
           // page is already in the memory, increment the counter
          map[pages[i]]++;
      } else {
           // page is not in the memory, remove the page with the lowest counter
          value  if (map.size() == frame_size) {  int min_counter_page = map.begin()-
          >first;  int min_counter = map.begin()->second;

              for (auto it = map.begin(); it != map.end(); it++)
                 {  if (it->second < min_counter) {
                 min_counter_page = it->first;  min_counter =
                 it->second;
                  }
              }

              map.erase(min_counter_page);
          }
          page_faults++;
      }

      // add the current page to the memory with counter value 1
     map[pages[i]] = 1;
    }

    return page_faults;
}

int main() {  int pages[] = {1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5};  int n = sizeof(pages) /
   sizeof(pages[0]);  int frame_size = 3;  int page_faults =
   lru_approximation_page_replacement(pages, n, frame_size);  cout << "Number of
   page faults using LRU-Approximation: " << page_faults << endl;

    return 0;
}
```

Output->



Output

```
/tmp/nT99LbaBQj.o
Number of page faults using LRU-Approximation: 8
```

FIFO:

This algorithm replaces the page that was first loaded into the main memory. The idea behind this algorithm is that the page that has been in the main memory for the longest period of time is less likely to be used again in the near future. The FIFO algorithm requires maintaining a queue of all the pages in the main memory, with the oldest page at the front of the queue and the newest

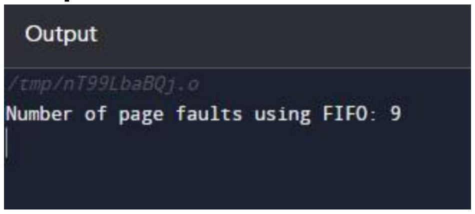
page at the back of the queue. When a page fault occurs, the page at the front of the queue is replaced with the new page.

Code:

```
#include <iostream>
#include <queue>

#include <unordered_set>

using namespace std;
```

```
int fifo_page_replacement(int pages[], int n, int frame_size) {
    queue<int> page_queue;

   unordered_set<int> page_set;

   int page_faults = 0;

    for (int i = 0; i < n; i++) {  if (page_set.find(pages[i]) ==
       page_set.end()) {  // page is not in the memory,
       remove the oldest page  if (page_set.size() ==
       frame_size) {  int oldest_page = page_queue.front();
       page_queue.pop();  page_set.erase(oldest_page);
           }
           page_faults++;
        }

        // add the current page to the memory
       page_queue.push(pages[i]);
       page_set.insert(pages[i]);
    }

    return page_faults;
}

int main() {  int pages[] = {1, 2, 3, 4, 1, 2, 5, 1,
   2, 3, 4, 5};  int n = sizeof(pages) /
   sizeof(pages[0]);  int frame_size = 3;
    int page_faults = fifo_page_replacement(pages, n, frame_size);

   cout << "Number of page faults using FIFO: " << page_faults <<

   endl;

    return 0;
}
```

Output->



```
/tmp/nT99LbaBQj.o
Number of page faults using FIFO: 9
```

OPTIMAL:

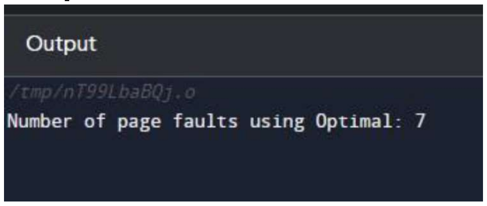
This algorithm replaces the page that will not be used for the longest period of time in the future. The idea behind this algorithm is to minimize the number of page faults that occur in the future. The optimal algorithm requires knowledge of the future memory references, which is not practical in most cases. However, it is often used as a benchmark to compare the performance of other page replacement algorithms.

CODE:

```
 #include <iostream>
 #include

<unordered_map>

#include <limits.h>  using

namespace std;

 int optimal_page_replacement(int pages[], int n, int frame_size) {
    unordered_map<int, int> map;
     int page_faults = 0;

     for (int i = 0; i < n; i++) {  if
        (map.find(pages[i]) != map.end()) {
        // page is already in the memory
         } else {
            // page is not in the memory, remove the page that will be used furthest in
the  future  if (map.size() == frame_size) {  int furthest_page = -1;  int
furthest_distance = INT_MIN;

                for (auto it = map.begin(); it != map.end(); it++)
                   {  int page = it->first;  int distance =
                   INT_MAX;
```

```
                    for (int j = i + 1; j < n; j++)
                      {  if (pages[j] == page) {
                      distance = j - i;  break;
                       }
                    }

                    if (distance > furthest_distance) {
                      furthest_page = page;
                      furthest_distance = distance;
                    }
                 }

                 map.erase(furthest_page);
              }
              page_faults++;
           }

           // add the current page to the memory
          map[pages[i]] = i;
       }

       return page_faults;
}
int main() {  int pages[] = {1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5};  int n =
   sizeof(pages) / sizeof(pages[0]);  int frame_size = 3;  int page_faults =
   optimal_page_replacement(pages, n, frame_size);  cout << "Number of
   page faults using Optimal: " << page_faults << endl;

    return 0;
}
```

Output->



```
/tmp/nT99LbaBQj.o
Number of page faults using Optimal: 7
```

10.FILE OPERATION SYSTEM CALLS:

CODE:

```
#include <iostream>
#include <fstream>  using
namespace std;

int main() {
  // Open a file in read mode  ifstream
  fin("example.txt");

  // Read the contents of the file
  string content;  getline(fin,
  content);  cout << content <<
  endl;

  // Close the file  fin.close();

  // Open a file in write mode
  ofstream fout("example.txt");

  // Write some content to the file  fout
  << "Hello World!" << endl;

  // Close the file  fout.close();

  // Open a file in append mode  ofstream
  fappend("example.txt", ios_base::app);

  // Append some content to the file  fappend << "This
  is some appended text." << endl;

  // Close the file  fappend.close();

  return 0;
}
```

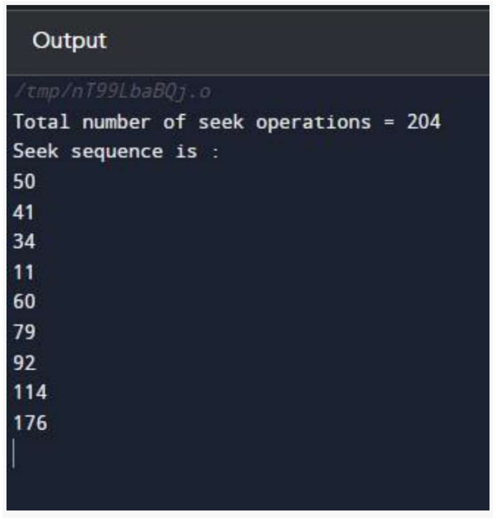
11. Disk Scheduling Algorithms.

Code: Shortest seek time first(SSTF)

```
#include <bits/stdc++.h> using
namespace std;
void calculatedifference(int request[], int head,
                         int diff[][2], int n)
{
    for(int i = 0; i < n; i++)
    {
        diff[i][0] = abs(head - request[i]);
    }
}

int findMIN(int diff[][2], int n)
{
    int index = -1;
    int minimum = 1e9;
    for(int i = 0; i< n; i++)
    {
        if (!diff[i][1] && minimum > diff[i][0])
        {
            minimum = diff[i][0];
            index = i;
        }
    }
    return index;
}

void shortestSeekTimeFirst(int request[],  int head, int n)
```

```
{
    if (n == 0)
    {
        return;
    }
    int diff[n][2] = { { 0, 0 } };

    int seekcount = 0;
    int seeksequence[n + 1] = {0};
    for(int i = 0; i < n; i++)
    {
        seeksequence[i] = head;
        calculatedifference(request, head, diff, n);
        int index = findMIN(diff,n);
        diff[index][1] = 1;
        seekcount += diff[index][0];
        head = request[index];
    }
    seeksequence[n] = head;
    cout << "Total number of seek operations = "
            << seekcount << endl;
    cout << "Seek sequence is : " << "\n";
    for(int i = 0; i <= n; i++)
    {
        cout << seeksequence[i] << "\n";
    }
}
int main()
{
    int n = 8;
    int proc[n] = { 176, 79, 34, 60, 92, 11, 41, 114 };
    shortestSeekTimeFirst(proc, 50, n);
    return 0;
}
```

Output->

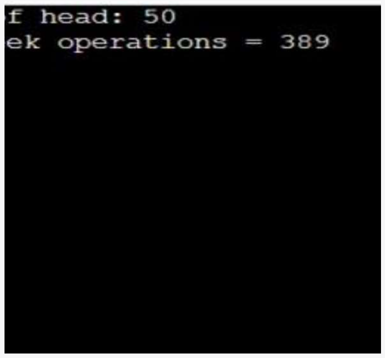

```
Output
/tmp/nT99LbaBQj.o
Total number of seek operations = 204
Seek sequence is :
50
41
34
11
60
79
92
114
176
```

Code: SCAN(Elevator) Algorithm

```
#include <bits/stdc++.h>
using namespace std;
int size = 8;  int
disk_size = 200;
void SCAN(int arr[], int head, string direction)
{
    int seek_count = 0;  int
   distance, cur_track;
   vector<int> left, right;
   vector<int> seek_sequence;
   if (direction == "left")
        left.push_back(0);
    else if (direction == "right")
   right.push_back(disk_size - 1);  for (int
   i = 0; i < size; i++) {
        if (arr[i] < head)  left.push_back(arr[i]);
        if (arr[i] > head)
            right.push_back(arr[i]);
    }
    std::sort(left.begin(), left.end());
   std::sort(right.begin(), right.end());
    int run = 2;
   while (run--) {
        if (direction == "left") {  for (int i =
           left.size() - 1; i >= 0; i--) {  cur_track =
           left[i];
                seek_sequence.push_back(cur_track);
               distance = abs(cur_track - head);
            seek_count += distance;
            head = cur_track;
            }
            direction = "right";
        }
        else if (direction == "right") {
            for (int i = 0; i < right.size(); i++) {  cur_track
           = right[i];
                    seek_sequence.push_back(cur_track);
            distance = abs(cur_track - head);
           seek_count += distance;
            head = cur_track;
            }
            direction = "left";
        }
    }
    cout << "Total number of seek operations = "
            << seek_count << endl;  cout << "Seek
   Sequence is" << endl;  for (int i = 0; i <
   seek_sequence.size(); i++) {
        cout << seek_sequence[i] << endl;
    }
}
int main()
{
    int arr[size] = { 176, 79, 34, 60,92, 11, 41, 114 };
                                                int head = 50;  string
      SCAN(arr, head, direction); direction = "left";
   return 0;
                  }
```

Code: C-SCAN (Circular Elevator) Disk Scheduling Algorithm

```
#include <bits/stdc++.h>
using namespace std;
int size = 8;  int
disk_size = 200;
void CSCAN(int arr[], int head)
{
     int seek_count = 0;  int
    distance, cur_track;
     vector<int> left, right;  vector<int>
    seek_sequence;  left.push_back(0);
    right.push_back(disk_size - 1);
     for (int i = 0; i < size; i++)
     {
          if (arr[i] < head)  left.push_back(arr[i]);
          if (arr[i] > head)  right.push_back(arr[i]);
     }
     std::sort(left.begin(), left.end());
    std::sort(right.begin(), right.end());
     for (int i = 0; i < right.size(); i++)
     {
          cur_track = right[i];
          seek_sequence.push_back(cur_track);  distance
         = abs(cur_track - head);  seek_count +=
         distance;
          head = cur_track;
     }
     head = 0;
     seek_count += (disk_size - 1);
     for (int i = 0; i < left.size(); i++)
     {
          cur_track = left[i];
          seek_sequence.push_back(cur_track);  distance
         = abs(cur_track - head);  seek_count +=
         distance;
          head = cur_track;
    }
    cout << "Total number of seek operations = "
         << seek_count << endl;  cout <<
   "Seek Sequence is" << endl;
    for (int i = 0; i < seek_sequence.size(); i++)
    {
        cout << seek_sequence[i] << endl;
    }
}
                                                                    int main()

        int arr[size] = {176, 79, 34, 60, 92, 11, 41, 114};
      int head = 50;
        cout << "Initial position of head: " << head << endl;
     CSCAN(arr, head);
    return 0;
```



```
f head: 50
ek operations = 389
```

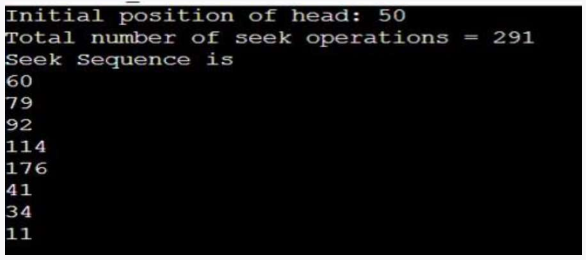
```
{



}
```

Code: LOOK Disk Scheduling Algorithm

```
int size = 8;
#include <bits/stdc++.h>
using namespace std; int
disk_size = 200;
void LOOK(int arr[], int head, string direction)
```

```
{
    int seek_count = 0;
    int distance, cur_track;
    vector<int> left, right;
    vector<int> seek_sequence;
    for (int i = 0; i < size; i++)
    {
        if (arr[i] < head)
            left.push_back(arr[i]);
        if (arr[i] > head)
            right.push_back(arr[i]);
    }
    std::sort(left.begin(), left.end());
    std::sort(right.begin(), right.end());

    int run = 2;
    while (run--)
    {
        if (direction == "left")
        {
            for (int i = left.size() - 1; i >= 0; i--)
            {
                cur_track = left[i];
                seek_sequence.push_back(cur_track);
                distance = abs(cur_track - head);
                seek_count += distance;
                head = cur_track;
            }
            direction = "right";
        }
        else if (direction == "right")
        {
            for (int i = 0; i < right.size(); i++)
            {
                cur_track = right[i];
                seek_sequence.push_back(cur_track);
                distance = abs(cur_track - head);
                seek_count += distance;
                head = cur_track;
            }
            direction = "left";
        }
    }
    cout << "Total number of seek operations = "
         << seek_count << endl;
    cout << "Seek Sequence is" << endl;
    for (int i = 0; i < seek_sequence.size(); i++)
    {
        cout << seek_sequence[i] << endl;
    }
}
int main()
{
    int arr[size] = {176, 79, 34, 60, 92, 11, 41, 114};
    int head = 50;
    string direction = "right";
    cout << "Initial position of head: "
         << head << endl;
    LOOK(arr, head, direction);
    return 0;
}
```

Output:



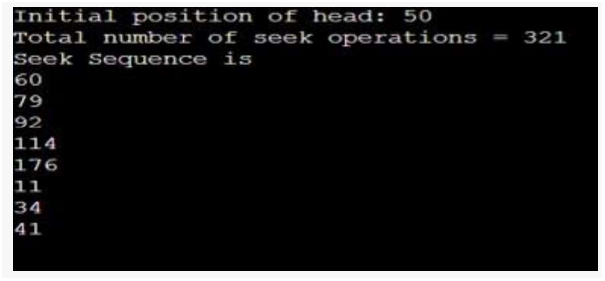
```
Initial position of head: 50
Total number of seek operations = 291
Seek Sequence is
60
79
92
114
176
41
34
11
```

Code: C-LOOK Disk Scheduling Algorithm

```
#include <bits/stdc++.h>
using namespace std;
int size = 8;  int
disk_size = 200;
void CLOOK(int arr[], int head)
{
    int seek_count = 0;  int
   distance, cur_track;
   vector<int> left, right;
   vector<int> seek_sequence;
    for (int i = 0; i < size; i++)
    {
        if (arr[i] < head)  left.push_back(arr[i]);
        if (arr[i] > head)
            right.push_back(arr[i]);
    }
    std::sort(left.begin(), left.end());
   std::sort(right.begin(), right.end());
    for (int i = 0; i < right.size(); i++)
    {
        cur_track = right[i];
        seek_sequence.push_back(cur_track);
       distance = abs(cur_track - head);  seek_count
       += distance;
        head = cur_track;
    }
    seek_count += abs(head - left[0]);  head =
   left[0];
    for (int i = 0; i < left.size(); i++)
    {
        cur_track = left[i];
        seek_sequence.push_back(cur_track);
       distance = abs(cur_track - head);  seek_count
       += distance;
        head = cur_track;
    }
```

```
     cout << "Total number of seek operations = "
          << seek_count << endl;  cout <<

}
int main()
{

    int arr[size] = {176, 79, 34, 60, 92, 11, 41, 114};
    int head = 50;
    cout << "Initial position of head: " << head << endl;
    CLOOK(arr, head);
    return 0;
}
         for (int i = 0; i < seek_sequence.size(); i++)
    {
          cout << seek_sequence[i] << endl;
    }

    "Seek Sequence is" << endl;
```

Output:

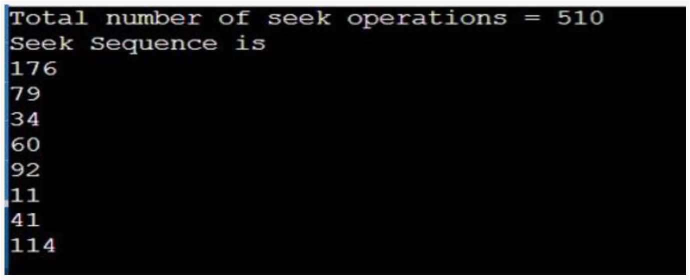


```
Initial position of head: 50
Total number of seek operations = 321
Seek Sequence is
60
79
92
114
176
11
34
41
```

Code: First Come First Serve Algorithm

```
#include <bits/stdc++.h>
using namespace std;
int size = 8;  void FCFS(int
arr[], int head)
{
    int seek_count = 0;  int
    distance, cur_track;
    for (int i = 0; i < size; i++)
    {
        cur_track = arr[i];  distance =
       abs(cur_track - head);  seek_count
       += distance;
        head = cur_track;
    }
   cout << "Total number of seek operations = " << seek_count << endl;
   cout << "Seek Sequence is" << endl;
   for (int i = 0; i < size; i++)
   {
       cout << arr[i] << endl;
   }
}
int main()
{
   int arr[size] = {176, 79, 34, 60, 92, 11, 41, 114};
   int head = 50;
   FCFS(arr, head);
   return 0;
}
```

Output:



```
Total number of seek operations = 510
Seek Sequence is
176
79
34
60
92
11
41
114
```