EXPERIMENT - 5

Aim: Write a program for page replacement policy using a)LRU b)FIFO c)Optimal

Theory:

1. LRU (Least Recently Used): This policy replaces the least recently used page when the system needs to bring in a new page that isn't currently in memory. The idea is to discard the page that has not been accessed for the longest period of time among all the currently resident pages.

2. FIFO (First-In, First-Out): This policy replaces the oldest page in memory, i.e., the one that has been in memory the longest. It operates on the principle that the page that has been in memory for the longest time is likely to be the least used or least needed in the near future.

3. Optimal: This is an idealized page replacement algorithm. It works by replacing the page that will not be used for the longest period of time in the future. Though it's considered the most efficient in theory, in practical systems, it's not implementable since it requires future knowledge of the pages to be accessed.

Implementing these algorithms involves maintaining data structures that keep track of the order or timestamps of page accesses, allowing the system to make informed decisions about which pages to replace when memory is full. These algorithms have their trade-offs in terms of simplicity, efficiency, and optimality in predicting future page accesses.

Source Code:

1. LRU:

```
#include <stdio.h>
int findLRU(int time[], int n) {
   int i, minimum = time[0], pos = 0;
   for (i = 1; i < n; ++i) {
      if (time[i] < minimum) {
         minimum = time[i];
         pos = i;
      }
   }
   return pos;
}
int main() {
   int no_of_frames, no_of_pages, frames[10], pages[30], counter = 0, time[10], flag1, flag2,
i, j,
                                          pos, faults = 0;
   printf("Enter number of frames: ");
   scanf("%d", &no_of_frames);
   printf("Enter number of pages: ");
```

```
    scanf("%d", &no_of_pages);
    printf("Enter reference string: ");
    for (i = 0; i < no_of_pages; ++i) scanf("%d", &pages[i]);
    for (i = 0; i < no_of_frames; ++i) frames[i] = -1;
    for (i = 0; i < no_of_pages; ++i) {
        flag1 = flag2 = 0;
        for (j = 0; j < no_of_frames; ++j) {
            if (frames[j] == pages[i]) {
                counter++;
                time[j] = counter;
                flag1 = flag2 = 1;
                break;
            }
        }
        if (flag1 == 0) {
            for (j = 0; j < no_of_frames; ++j) {
                if (frames[j] == -1) {
                    counter++;
                    faults++;
                    frames[j] = pages[i];
                    time[j] = counter;
                    flag2 = 1;
                    break;
                }
            }
        }
        if (flag2 == 0) {
            pos = findLRU(time, no_of_frames);
            counter++;
            faults++;
            frames[pos] = pages[i];
            time[pos] = counter;
        }
        printf("\n");
        for (j = 0; j < no_of_frames; ++j) printf("%d\t", frames[j]);
    }
    printf("\nTotal Page Faults = %d", faults);
    return 0;
}
```

2. FIFO:

```
#include <stdio.h>
int main() {
    int incomingStream[] = {4, 1, 2, 4, 5}, frames = 3,pageFaults = 0,m, n, s, pages;
    pages = sizeof(incomingStream) / sizeof(incomingStream[0]);
    printf("Incoming \t Frame 1 \t Frame 2 \t Frame 3");
```

```
    int temp[frames];
    for (m = 0; m < frames; m++) temp[m] = -1;
    for (m = 0; m < pages; m++) {
        s = 0;
        for (n = 0; n < frames; n++) {
            if (incomingStream[m] == temp[n]) {
                s++;
                pageFaults--;
            }
        }
        pageFaults++;
        if ((pageFaults <= frames) && (s == 0)) temp[m] = incomingStream[m];
        else if (s == 0) temp[(pageFaults - 1) % frames] = incomingStream[m];
        printf("\n%d\t\t\t", incomingStream[m]);
        for (n = 0; n < frames; n++) {
            if (temp[n] != -1)  printf(" %d\t\t\t", temp[n]);
            else printf(" - \t\t\t");
        }
    }
    printf("\nTotal Page Faults:\t%d\n", pageFaults);
    return 0;
}
```

3. Optimal:

```
#include <stdio.h>
// This function checks if current strea item(key) exists in any of the frames or not
int search(int key, int frame_items[], int frame_occupied) {
    for (int i = 0; i < frame_occupied; i++)
        if (frame_items[i] == key) return 1;
    return 0;
}
void printOuterStructure(int max_frames) {
    printf("Stream ");
    for (int i = 0; i < max_frames; i++)
        printf("Frame%d ", i + 1);
}
void printCurrFrames(int item, int frame_items[], int frame_occupied, int max_frames) {
    printf("\n%d \t\t", item);
    for (int i = 0; i < max_frames; i++) {
        if (i < frame_occupied) printf("%d \t\t", frame_items[i]);
        else printf("- \t\t");
    }
}
int predict(int ref_str[], int frame_items[], int refStrLen, int index, int frame_occupied) {
    int result = -1, farthest = index;
    for (int i = 0; i < frame_occupied; i++) {
        int j;
```

```
        for (j = index; j < refStrLen; j++) {
            if (frame_items[i] == ref_str[j]) {
                if (j > farthest) {
                    farthest = j;
                    result = i;
                }
                break;
            }
        }
        if (j == refStrLen) return i;
    }
    return (result == -1) ? 0 : result;
}
void optimalPage(int ref_str[], int refStrLen, int frame_items[], int max_frames) {
    int frame_occupied = 0;
    printOuterStructure(max_frames);
    int hits = 0;
    for (int i = 0; i < refStrLen; i++) {
        if (search(ref_str[i], frame_items, frame_occupied)) {
            hits++;
            printCurrFrames(ref_str[i], frame_items, frame_occupied, max_frames);
            continue;
        }
        if (frame_occupied < max_frames) {
            frame_items[frame_occupied] = ref_str[i];
            frame_occupied++;
            printCurrFrames(ref_str[i], frame_items, frame_occupied, max_frames);
        }
        else {
            int pos = predict(ref_str, frame_items, refStrLen, i + 1, frame_occupied);
            frame_items[pos] = ref_str[i];
            printCurrFrames(ref_str[i], frame_items, frame_occupied, max_frames);
        }
    }
    printf("\n\nHits: %d\n", hits);
    printf("Misses: %d", refStrLen - hits);
}
int main() {
    int ref_str[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1};
    int refStrLen = sizeof(ref_str) / sizeof(ref_str[0]);
    int max_frames = 3,  frame_items[max_frames];
    optimalPage(ref_str, refStrLen, frame_items, max_frames);
    return 0;
}
```

OUTPUT:

1. LRU:

```
Enter number of frames: 3
Enter number of pages: 10
Enter reference string: 7 5 9 4 3 7 9 6 2 1

7       -1      -1
7       5       -1
7       5       9
4       5       9
4       3       9
4       3       7
9       3       7
9       6       7
9       6       2
1       6       2
```

2. FIFO:

```
Incoming        Frame 1         Frame 2         Frame 3
4                       4                       -                       -
1                       4                       1                       -
2                       4                       1                       2
4                       4                       1                       2
5                       5                       1                       2
Total Page Faults:      4
```

3. Optimal:

```
Stream Frame1 Frame2 Frame3
7               7               -               -
0               7               0               -
1               7               0               1
2               2               0               1
0               2               0               1
3               2               0               3
0               2               0               3
4               2               4               3
2               2               4               3
3               2               4               3
0               2               0               3
3               2               0               3
2               2               0               3
1               2               0               1
2               2               0               1
0               2               0               1
1               2               0               1
7               7               0               1
0               7               0               1
1               7               0               1

Hits: 11
Misses: 9
```

EXPERIMENT - 6

Aim: Write a program to implement first fit, best fit and worst fit algorithm for memory management.

Theory :

First-Fit Allocation is a memory allocation technique used in operating systems to allocate memory to a process. In First-Fit, the operating system searches through the list of free blocks of memory, starting from the beginning of the list, until it finds a block that is large enough to accommodate the memory request from the process. Once a suitable block is found, the operating system splits the block into two parts: the portion that will be allocated to the process, and the remaining free block.

Best-Fit Allocation is a memory allocation technique used in operating systems to allocate memory to a process. In Best-Fit, the operating system searches through the list of free blocks of memory to find the block that is closest in size to the memory request from the process. Once a suitable block is found, the operating system splits the block into two parts: the portion that will be allocated to the process, and the remaining free block.

Worst-fit allocation is a memory management strategy that operating systems use to allocate memory resources to processes. It involves finding and allocating the largest available block of memory to the process in question. In other words, the worst-fit algorithm looks for the block of memory that is furthest in size from the requested size of the process. The aim of worst-fit allocation algorithm is to maximize memory fragmentation. It keeps the largest blocks available, which can be useful in situations where larger processes may be allocated in the future. However, it can result in inefficient memory utilization, as smaller blocks may not be fully utilized, leading to fragmentation.

Source Code:

1. FIRST FIT:

```
#include <stdio.h>
void main() {
   int bsize[10], psize[10], bno, pno, flags[10], allocation[10], i, j;
   for (i = 0; i < 10; i++) {
      flags[i] = 0;
      allocation[i] = -1;
   }
   printf("Enter no. of blocks: ");
   scanf("%d", &bno);
   printf("\nEnter size of each block: ");
```

```
    for (i = 0; i < bno; i++) scanf("%d", &bsize[i]);
    printf("\nEnter no. of processes: ");
    scanf("%d", &pno);
    printf("\nEnter size of each process: ");
    for (i = 0; i < pno; i++) scanf("%d", &psize[i]);
    for (i = 0; i < pno; i++) // allocation as per first fit
        for (j = 0; j < bno; j++)
            if (flags[j] == 0 && bsize[j] >= psize[i]) {
                allocation[j] = i;
                flags[j] = 1;
                break;
            }
    // display allocation details
    printf("\nBlock no.\tsize\t\tprocess no.\t\tsize");
    for (i = 0; i < bno; i++) {
        printf("\n%d\t\t%d\t\t", i + 1, bsize[i]);
        if (flags[i] == 1) printf("%d\t\t\t%d", allocation[i] + 1, psize[allocation[i]]);
        else printf("Not allocated");
    }
}
```

2. BEST FIT:

```
#include <stdio.h>
void main() {
    int fragment[20], b[20], p[20], i, j, nb, np, temp, lowest = 9999;
    static int barray[20], parray[20];
    printf("\n\t\t\tMemory Management Scheme - Best Fit");
    printf("\nEnter the number of blocks:");
    scanf("%d", &nb);
    printf("Enter the number of processes:");
    scanf("%d", &np);
    printf("\nEnter the size of the blocks:-\n");
    for (i = 1; i <= nb; i++) {
        printf("Block no.%d:", i);
        scanf("%d", &b[i]);
    }
    printf("\nEnter the size of the processes :-\n");
    for (i = 1; i <= np; i++) {
        printf("Process no.%d:", i);
        scanf("%d", &p[i]);
    }
    for (i = 1; i <= np; i++) {
        for (j = 1; j <= nb; j++) {
            if (barray[j] != 1) {
```

```
            temp = b[j] - p[i];
            if (temp >= 0)
               if (lowest > temp) {
                  parray[i] = j;
                  lowest = temp;
               }
         }
      }
      fragment[i] = lowest;
      barray[parray[i]] = 1;
      lowest = 10000;
   }
   printf("\nProcess_no\tProcess_size\tBlock_no\tBlock_size\tFragment");
   for (i = 1; i <= np && parray[i] != 0; i++)
      printf("\n%d\t\t%d\t\t%d\t\t%d\t\t%d", i, p[i], parray[i], b[parray[i]], fragment[i]);
}
```

3. WORST FIT:

```
#include <stdio.h>
void implimentWorstFit(int blockSize[], int blocks, int processSize[], int processes) {
   // This will store the block id of the allocated block to a process
   int allocation[processes];
   int occupied[blocks];
   for (int i = 0; i < processes; i++) allocation[i] = -1;
   for (int i = 0; i < blocks; i++) occupied[i] = 0;
   for (int i = 0; i < processes; i++) {
      int indexPlaced = -1;
      for (int j = 0; j < blocks; j++) {
         if (blockSize[j] >= processSize[i] && !occupied[j]) {
            if (indexPlaced == -1) indexPlaced = j;
            else if (blockSize[indexPlaced] < blockSize[j]) indexPlaced = j;
         }
      }
       if (indexPlaced != -1) {
         allocation[i] = indexPlaced;
         occupied[indexPlaced] = 1 :
         blockSize[indexPlaced] -= processSize[i];
      }
   }
   printf("\nProcess No.\tProcess Size\tBlock no.\n");
   for (int i = 0; i < processes; i++) {
      printf("%d \t\t\t %d \t\t\t", i + 1, processSize[i]);
      if (allocation[i] != -1) printf("%d\n", allocation[i] + 1);
      else printf("Not Allocated\n");
```

```
    }
}
// Driver code
int main() {
    int blockSize[] = {100, 50, 30, 120, 35};
    int processSize[] = {40, 10, 30, 60};
    int blocks = sizeof(blockSize) / sizeof(blockSize[0]);
    int processes = sizeof(processSize) / sizeof(processSize[0]);
    implimentWorstFit(blockSize, blocks, processSize, processes);
    return 0;
}
```

OUTPUTS:

1. First Fit

```
[ec2-user@ip-172-31-23-10 ~]$ vi firstfit.c
[ec2-user@ip-172-31-23-10 ~]$ gcc firstfit.c -o ff
[ec2-user@ip-172-31-23-10 ~]$ ./ff
Enter no. of blocks: 3

Enter size of each blocks: 12
7
4

Enter no. of processes: 3

Enter size of each process: 7
4
9

Block no.       size            process no.             size
1               12              1                       7
2               7               2                       4
3               4               Not allocated[ec2-user@ip-172-31-23-10 ~]$
```

2. Best Fit

```
[ec2-user@ip-172-31-23-10 ~]$ vi bestfit.c
[ec2-user@ip-172-31-23-10 ~]$ gcc bestfit.c -o bf
[ec2-user@ip-172-31-23-10 ~]$ ./bf

                        Memory Management Schema - Best Fir
Enter the number of blocks:5
Enter the number of processes:4

Enter the size of the blocks:-
Blocks no.1:10
Blocks no.2:15
Blocks no.3:5
Blocks no.4:9
Blocks no.5:3

Enter the size of the processes :-
Process no.1:1
Process no.2:4
Process no.3:7

Process_no      Process_size    Block_no        Block_size      Fragment
1               1               5               3               2
2               4               3               5               1
3               7               4               9               2[ec2-user@ip-172-31-23-10 ~]
$
```

3. Worst Fit

```
[ec2-user@ip-172-31-23-10 ~]$ vi wordtfit.c
[ec2-user@ip-172-31-23-10 ~]$ gcc wordtfit.c -o wf
[ec2-user@ip-172-31-23-10 ~]$ ./wf

Process No.     Process Size    Block No.
1                       40                      4
2                       10                      1
3                       30                      2
4                       60                      Not Allocated
```

EXPERIMENT - 7

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

Theory:

The reader/writer problem is a classic synchronization issue in computer science where multiple processes (readers and writers) concurrently access a shared resource, such as a file or a database. The goal is to ensure that multiple readers can access the resource simultaneously for reading, but exclusive access is granted to only one writer at a time to perform writing operations. This prevents data inconsistencies that may arise when a writer modifies the resource while it is being read by a reader.

In this scenario, semaphores are synchronization primitives used to control access to the shared resource. Semaphores can have different values, typically initialized to restrict access to the resource. For instance, a semaphore can be used to track the number of readers currently accessing the resource. When a reader wishes to read, it checks the semaphore value. If it's permissible (not locked by a writer), the reader increments the semaphore value, allowing multiple readers to read concurrently. However, when a writer wants to write, it checks and locks the semaphore, ensuring exclusive access to modify the resource while preventing other readers or writers from accessing it.

Implementing the reader/writer problem using semaphores involves careful synchronization and coordination among readers and writers through the use of these semaphore-based mechanisms, ensuring data integrity and avoiding race conditions during concurrent access to the shared resource.

Source Code:

```
#include <iostream>
#include <pthread.h>
#include <unistd.h>

using namespace std;

class monitor {

private:
int rcnt, waitr,wcn,waitw;
pthread_cond_t canread;
pthread_cond_t canwrite;
pthread_mutex_t condlock;
```

```
public:
monitor() {
        rcnt = 0; wcnt = 0; waitr = 0; waitw = 0;
        pthread_cond_init(&canread, NULL);
        pthread_cond_init(&canwrite, NULL);
        pthread_mutex_init(&condlock, NULL);
}

void beginread(int i) {
        pthread_mutex_lock(&condlock);
        if (wcnt == 1 || waitw > 0) {
                waitr++;
                pthread_cond_wait(&canread, &condlock);
                waitr--;
        }
        rcnt++;
        cout << "reader " << i << " is reading\n";
        pthread_mutex_unlock(&condlock);
        pthread_cond_broadcast(&canread);
}

void endread(int i) {
        pthread_mutex_lock(&condlock);
        if (--rcnt == 0) pthread_cond_signal(&canwrite);
        pthread_mutex_unlock(&condlock);
}

void beginwrite(int i) {
        pthread_mutex_lock(&condlock);
        if (wcnt == 1 || rcnt > 0) {
                ++waitw;
                pthread_cond_wait(&canwrite, &condlock);
                --waitw;
        }
        wcnt = 1;
        cout << "writer " << i << " is writing\n";
        pthread_mutex_unlock(&condlock);
}

void endwrite(int i) {
        pthread_mutex_lock(&condlock);
        wcnt = 0;
        if (waitr > 0) pthread_cond_signal(&canread);
        else pthread_cond_signal(&canwrite);
```

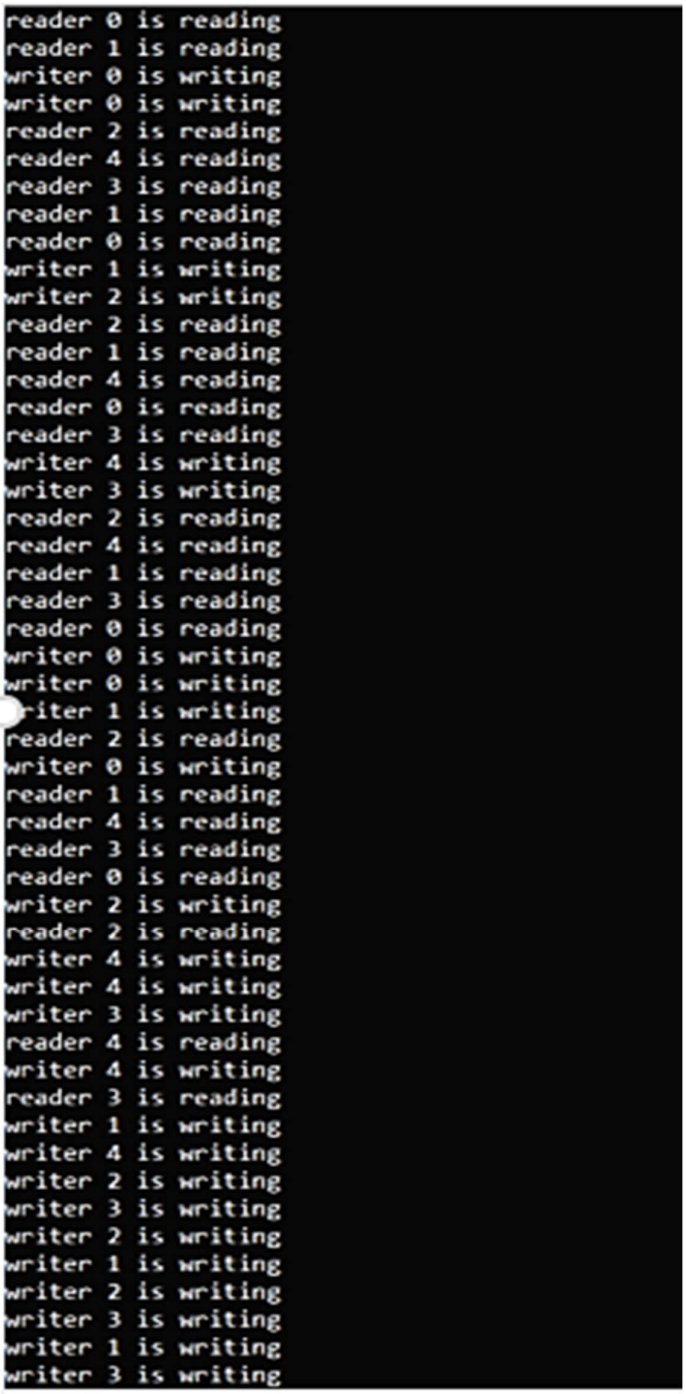
```
        pthread_mutex_unlock(&condlock);
        }
}
M;

void* reader(void* id) {
        int c = 0;
        int i = *(int*)id;
        while (c < 5) {
                usleep(1);
                M.beginread(i);
                M.endread(i);
                c++;
        }
}

void* writer(void* id) {
        int c = 0;
        int i = *(int*)id;
        while (c < 5) {
                usleep(1);
                M.beginwrite(i);
                M.endwrite(i);
                c++;
        }
}

int main() {
        pthread_t r[5], w[5];
        int id[5];
        for (int i = 0; i < 5; i++) {
                id[i] = i;
                pthread_create(&r[i], NULL, &reader, &id[i]);
                pthread_create(&w[i], NULL, &writer, &id[i]);
        }
        for (int i = 0; i < 5; i++) {
                pthread_join(r[i], NULL); pthread_join(w[i], NULL);
        }
}
```

OUTPUT:


reader 0 is reading
reader 1 is reading
writer 0 is writing
writer 0 is writing
reader 2 is reading
reader 4 is reading
reader 3 is reading
reader 1 is reading
reader 0 is reading
writer 1 is writing
writer 2 is writing
reader 2 is reading
reader 1 is reading
reader 4 is reading
reader 0 is reading
reader 3 is reading
writer 4 is writing
writer 3 is writing
reader 2 is reading
reader 4 is reading
reader 1 is reading
reader 3 is reading
reader 0 is reading
writer 0 is writing
writer 0 is writing
writer 1 is writing
reader 2 is reading
writer 0 is writing
reader 1 is reading
reader 4 is reading
reader 3 is reading
reader 0 is reading
writer 2 is writing
reader 2 is reading
writer 4 is writing
writer 4 is writing
writer 3 is writing
reader 4 is reading
writer 4 is writing
reader 3 is reading
writer 1 is writing
writer 4 is writing
writer 2 is writing
writer 3 is writing
writer 2 is writing
writer 1 is writing
writer 2 is writing
writer 3 is writing
writer 1 is writing
writer 3 is writing

EXPERIMENT - 8

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

Theory:

The Producer-Consumer problem revolves around managing a shared, bounded buffer between two types of processes: producers and consumers. Producers generate data items and place them into a shared buffer, while consumers retrieve and process these items from the buffer. The challenge is to ensure synchronization and avoid issues like race conditions, buffer overflow, or underflow.

Semaphores are employed to regulate access to the shared buffer. Two semaphores are often used: one to control the number of empty slots in the buffer (indicating available space for producers to place items), and another to manage the number of filled slots in the buffer (indicating items ready for consumers to retrieve).

When a producer wishes to add an item to the buffer, it checks the empty slots semaphore. If there's space available (semaphore value > 0), it decrements the empty slots semaphore (indicating consumption of an empty slot) and proceeds to add the item to the buffer. After the addition, the producer increments the filled slots semaphore to signal that there's a new item for the consumers.

On the other hand, when a consumer wants to retrieve an item from the buffer, it checks the filled slots semaphore. If there are items available (semaphore value > 0), it decrements the filled slots semaphore (indicating consumption of a filled slot), retrieves the item from the buffer, and then increments the empty slots semaphore to signal the availability of an empty slot for producers.

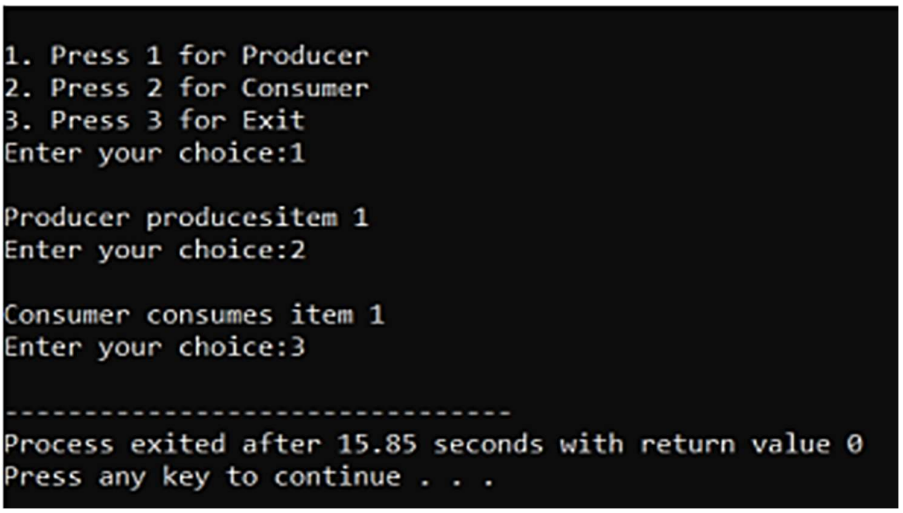
This synchronization mechanism ensures that producers and consumers coordinate their access to the shared buffer, preventing issues like buffer overflow or underflow while allowing multiple producers and consumers to work concurrently.

Implementing the Producer-Consumer problem using semaphores involves careful handling of these semaphores to control access to the shared buffer, ensuring that producers and consumers properly coordinate their actions to avoid conflicts and maintain the integrity of the data within the buffer.

Source Code:

```
#include <stdio.h>
#include <stdlib.h>
int mutex = 1;
int full = 0;
int empty = 10, x = 0;
void producer() {
        --mutex;        ++full; --empty;        x++;
        printf("\nProducer produces" "item %d", x);
        ++mutex;
}
void consumer() {
--mutex;        --full;  ++empty;
printf("\nConsumer consumes " "item %d", x);
x--;
++mutex;
}
int main() {
        int n, i;
        printf("\n1. Press 1 for Producer \n2. Press 2 for Consumer \n3. Press 3 for Exit");
        #pragma omp critical
        for (i = 1; i > 0; i++) {
                printf("\nEnter your choice:");
                scanf("%d", &n);
                switch (n) {
                        case 1:
                                if ((mutex == 1) && (empty != 0)) producer();
                                else printf("Buffer is full!");
                                break;
                        case 2:
                                if ((mutex == 1) && (full != 0)) consumer();
                                else printf("Buffer is empty!");
                                break;
                        case 3:
                                exit(0);
                        break;
                }
        }
}
```

OUTPUT:


1. Press 1 for Producer
2. Press 2 for Consumer
3. Press 3 for Exit
Enter your choice:1
Producer producesitem 1
Enter your choice:2
Consumer consumes item 1
Enter your choice:3

Process exited after 15.85 seconds with return value 0
Press any key to continue . . .

EXPERIMENT - 9

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

Theory:

The Banker's Algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for the predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.

Advantages: Following are the essential characteristics of the Banker's algorithm:

1. It contains various resources that meet the requirements of each process.
2. Each process should provide information to the operating system for upcoming resource requests, the number of resources, and how long the resources will be held.
3. It helps the operating system manage and control process requests for each type of resource in the computer system.
4. The algorithm has a Max resource attribute that represents indicates each process can hold the maximum number of resources in a system.

Disadvantages:

1. It requires a fixed number of processes, and no additional processes can be started in the system while executing the process.
2. The algorithm does no longer allows the processes to exchange its maximum needs while processing its tasks.
3. Each process has to know and state their maximum resource requirement in advance for the system.
4. The number of resource requests can be granted in a finite time, but the time limit for allocating the resources is one year.

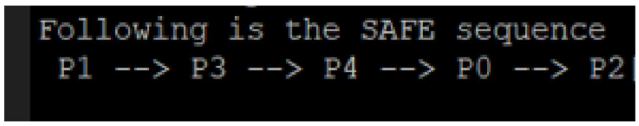
Source Code:

```
#include <stdio.h>
int main()
{
   // P0, P1, P2, P3, P4 are the Process names here
   int n, m, i, j, k;
   n = 5;                              // Number of processes
   m = 3;                              // Number of resources
   int alloc[5][3] = {{0, 1, 0},       // P0 // Allocation Matrix
                      {2, 0, 0},  // P1
                      {3, 0, 2},  // P2
                      {2, 1, 1},  // P3
                      {0, 0, 2}}; // P4
   int max[5][3] = {{7, 5, 3},         // P0 // MAX Matrix
```

```
            {3, 2, 2},          // P1
            {9, 0, 2},          // P2
            {2, 2, 2},          // P3
            {4, 3, 3}};         // P4
int avail[3] = {3, 3, 2};                   // Available Resources
int f[n], ans[n], ind = 0;
for (k = 0; k < n; k++)  f[k] = 0;
int need[n][m];
for (i = 0; i < n; i++)
   for (j = 0; j < m; j++) need[i][j] = max[i][j] - alloc[i][j];
int y = 0;
for (k = 0; k < 5; k++) {
   for (i = 0; i < n; i++)  {
      if (f[i] == 0) {
         int flag = 0;
         for (j = 0; j < m; j++)  {
            if (need[i][j] > avail[j])  {
               flag = 1;
               break;
            }
         }
         if (flag == 0) {
            ans[ind++] = i;
            for (y = 0; y < m; y++)
               avail[y] += alloc[i][y];
            f[i] = 1;
         }
      }
   }
}
int flag = 1;
for (int i = 0; i < n; i++) {
   if (f[i] == 0) {
      flag = 0;
      printf("The following system is not safe");
      break;
   }
}
if (flag == 1) {
   printf("Following is the SAFE Sequence\n");
```

```
    for (i = 0; i < n - 1; i++) printf(" P%d ->", ans[i]);
    printf(" P%d", ans[n - 1]);
  }
  return (0);
}
```

OUTPUT:



EXPERIMENT - 10

Aim: Write C programs to implement the various File Organization Techniques- a) Indexed File Organisation b) Heap File Organization c) Hash File Organization d) Sequential File Organization

Theory:

Below are brief explanations of each file organization technique along with a high-level overview of implementing:

a) Indexed File Organization:

Indexed file organization involves using an index structure to facilitate faster access to records in a file. The index holds pointers or references to actual records.
Implementation in C: To implement indexed file organization in C, you'd create a structure for the index table that contains key fields and pointers to record positions in the main file. Additionally, you'd manage file I/O operations to create, update, and use the index while performing insertions, deletions, and searches.

b) Heap File Organization:

Heap file organization doesn't have a specific order for storing records; instead, records are inserted anywhere there's free space in the file.
Implementation in C: Implementing a heap file involves managing free space within the file. You'd design a structure for records, handle file I/O operations to insert records at available locations, and manage pointers or metadata to track free space and record positions efficiently.

c) Hash File Organization:

Hash file organization uses a hash function to directly map keys to specific locations in the file, enabling fast access to records.
Implementation in C: To implement a hashed file in C, you'd design a hash function and create a hash table that maps hashed keys to record positions in the file. You'd also handle collision resolution strategies, manage file I/O operations, and perform efficient storage and retrieval based on hashed keys.

d) Sequential File Organization:

Sequential file organization stores records in consecutive order based on their insertion sequence.
Implementation in C: Implementing a sequential file involves defining a structure for records and handling file I/O operations using functions like fopen, fwrite, fread, and fclose to sequentially write and read records in the file. Retrieval typically involves iterating through the file to find specific records based on some criteria.

Source Code:

a) Indexed File Organization:

```
#include <iostream>
#include <fstream>
#include <cstring>

using namespace std;

struct Record {
   int key;
   char data[50];
};

int main() {
   ofstream indexFile("indexed_file.dat", ios::binary);
   ofstream dataFile("indexed_data.dat", ios::binary);

   // Sample records
   Record records[] = {{1, "John"}, {2, "Doe"}, {3, "Alice"}};

   for (const auto& record : records) {
      // Writing to data file
      dataFile.write(reinterpret_cast<const char*>(&record), sizeof(Record));

      // Writing to index file
      indexFile.write(reinterpret_cast<const char*>(&record.key), sizeof(int));
      long pos = dataFile.tellp();  // Get the position of the record in data file
      indexFile.write(reinterpret_cast<const char*>(&pos), sizeof(long));
   }

   indexFile.close();
   dataFile.close();

   return 0;
}
```

b) Heap File Organization:

```
#include <iostream>
#include <fstream>
#include <cstring>

using namespace std;

struct Record {
   int key;
   char data[50];
};

int main() {
   ofstream heapFile("heap_file.dat", ios::binary);

   // Sample records
   Record records[] = {{1, "John"}, {2, "Doe"}, {3, "Alice"}};

   for (const auto& record : records) {
      heapFile.write(reinterpret_cast<const char*>(&record), sizeof(Record));
   }

   heapFile.close();

   return 0;
}
```

c) Hash File Organization:

```
#include <iostream>
#include <fstream>
#include <cstring>

using namespace std;

struct Record {
   int key;
   char data[50];
};

const int HASH_TABLE_SIZE = 10;

int main() {
   ofstream hashFile[HASH_TABLE_SIZE];

   // Create separate files for each bucket
   for (int i = 0; i < HASH_TABLE_SIZE; ++i) {
      string fileName = "hash_file_" + to_string(i) + ".dat";
      hashFile[i].open(fileName, ios::binary);
   }

   // Sample records
   Record records[] = {{1, "John"}, {2, "Doe"}, {3, "Alice"}};

   for (const auto& record : records) {
      // Determine the hash bucket using a simple hash function
      int bucket = record.key % HASH_TABLE_SIZE;

      // Writing to the corresponding hash file
      hashFile[bucket].write(reinterpret_cast<const char*>(&record), sizeof(Record));
   }

   // Close all hash files
   for (int i = 0; i < HASH_TABLE_SIZE; ++i) {
      hashFile[i].close();
   }

   return 0;
}
```

d) Sequential File Organization:

```
#include <iostream>
#include <fstream>
#include <cstring>

using namespace std;

struct Record {
   int key;
   char data[50];
};

int main() {
   ofstream sequentialFile("sequential_file.dat", ios::binary);

   // Sample records
   Record records[] = {{1, "John"}, {2, "Doe"}, {3, "Alice"}};

   for (const auto& record : records) {
      sequentialFile.write(reinterpret_cast<const char*>(&record), sizeof(Record));
   }

   sequentialFile.close();

   return 0;
}
```

VIVA QUESTIONS:

Q1. What is CPU scheduling, and why is it necessary in an operating system?
Answer: CPU scheduling is the process by which the operating system selects processes from the ready queue and allocates the CPU to them. It's necessary to maximize CPU utilization, throughput, and fairness among processes.

Q2. Explain the First Come First Serve (FCFS) CPU scheduling algorithm. How does it work?
Answer: FCFS is a non-preemptive scheduling algorithm where processes are executed in the order they arrive in the ready queue. The first process that arrives is the first to be executed, and subsequent processes are scheduled in the order of arrival.

Q3. What is the waiting time and turnaround time in the context of CPU scheduling? How are they calculated in FCFS?
Answer: Waiting time is the total time a process spends waiting in the ready queue before getting CPU time. Turnaround time is the total time from the submission of the process to its completion. In FCFS, waiting time for a process is the sum of waiting times of all previous processes, and turnaround time is waiting time plus burst time.

Q4. Can you describe a scenario where FCFS scheduling may lead to poor performance?
Answer: FCFS may lead to poor performance in scenarios with a mix of short and long burst time processes. If a long process arrives first, short processes have to wait a long time, leading to increased waiting times and reduced throughput.

Q5. What is the average waiting time in FCFS scheduling? Can you provide a formula for its calculation?
Answer: The average waiting time in FCFS is the sum of waiting times for all processes divided by the number of processes. The formula is:

Average Waiting Time = (Σ Waiting Time) / Number of Processes.

Q6. What are the advantages and disadvantages of using FCFS scheduling?
Answer:
Advantages: Simple and easy to understand.
Disadvantages: Poor performance in terms of waiting time and throughput, especially with a mix of short and long processes.

VIVA QUESTIONS:

Q1. What is Shortest Job First (SJF) CPU scheduling, and how does it differ from FCFS?
Answer: SJF is a scheduling algorithm where the process with the shortest burst time is scheduled first. Unlike FCFS, it prioritizes shorter jobs, aiming to minimize waiting times.

Q2. Explain the difference between preemptive and non-preemptive SJF scheduling.
Answer: In preemptive SJF, a new process with a shorter burst time can preempt the currently running process. In non-preemptive SJF, the scheduler waits until the current process completes before selecting the next one.

Q3. What is the role of the ready queue in SJF scheduling?
Answer: The ready queue holds processes that are ready to be executed. In SJF, the ready queue is ordered by burst time, with the process having the shortest burst time at the front.

Q4. How does SJF scheduling handle the situation where two processes have the same burst time?
Answer: SJF scheduling may use additional criteria, such as arrival time or process ID, to break ties when two processes have the same burst time.

Q5. Can you describe a scenario where SJF scheduling may lead to poor performance?
Answer: SJF may lead to poor performance in scenarios where the burst times are not accurately predicted, causing longer processes to be delayed if shorter processes arrive later.

Q6. What are the advantages and disadvantages of using SJF scheduling?
Answer:
Advantages: Efficient in minimizing waiting times for shorter processes.
Disadvantages: Difficult to accurately predict burst times, may cause starvation for longer processes.

VIVA QUESTIONS:

Q1. What is Priority Scheduling, and how does it differ from SJF and FCFS?
Answer: Priority Scheduling is a scheduling algorithm where each process is assigned a priority, and the process with the highest priority is scheduled first. Unlike SJF and FCFS, it doesn't solely consider burst time but prioritizes based on a predefined priority value.

Q2. How are priorities assigned to processes in Priority Scheduling?
Answer: Priorities can be assigned based on various criteria, such as the process type, importance, deadlines, or any other metric relevant to the system's requirements.

Q3. What is the role of the ready queue in Priority Scheduling?
Answer: The ready queue holds processes that are ready to be executed. In Priority Scheduling, the process with the highest priority is selected from the ready queue for execution.

Q4. How does Priority Scheduling handle the situation where two processes have the same priority?
Answer: Priority Scheduling may use additional criteria, such as arrival time or process ID, to break ties when two processes have the same priority.

Q5. Can you describe a scenario where Priority Scheduling may lead to poor performance?
Answer: Priority Scheduling may lead to poor performance if the priorities are not assigned appropriately, causing important processes to wait for lower-priority processes.

Q6. What are the advantages and disadvantages of using Priority Scheduling?
Answer:
Advantages: Allows prioritization of important processes.
Disadvantages: May lead to starvation for lower-priority processes.

VIVA QUESTIONS:

Q1. What is Round Robin (RR) CPU scheduling, and how does it differ from other scheduling algorithms like FCFS and SJF?

Answer: Round Robin is a preemptive scheduling algorithm where each process is assigned a fixed time slice (quantum) to execute. It differs from FCFS and SJF by allowing processes to share the CPU in a time-sliced manner.

Q2. Explain the concept of time quantum in Round Robin scheduling. How is it determined?

Answer: The time quantum is the maximum amount of time a process is allowed to run in a single burst before being preempted. It's determined based on the system's requirements and the need for a balance between responsiveness and context switch overhead.

Q3. How does Round Robin handle situations where a process doesn't complete its execution within the time quantum?

Answer: If a process doesn't complete within the time quantum, it is preempted, and the next process in the ready queue is given a chance to execute. The unfinished process is placed back in the ready queue.

Q4. Can you describe a scenario where Round Robin scheduling may lead to poor performance?

Answer: Round Robin may lead to poor performance when the time quantum is too large, causing processes to wait longer for their turn, or when there are mostly long processes in the system.

Q5. What is the significance of the time quantum in determining the efficiency of Round Robin scheduling?

Answer: The time quantum determines the balance between responsiveness and context switch overhead. A smaller time quantum provides better responsiveness but increases context switch overhead, while a larger time quantum reduces context switch overhead but may result in longer waiting times.

Q6. What are the advantages and disadvantages of using Round Robin scheduling?

Answer:
Advantages: Fair allocation of CPU time, avoids starvation.
Disadvantages: Overhead due to frequent context switches, may not perform well with a mix of short and long processes.

VIVA QUESTIONS:

Q1. What is page replacement, and why is it necessary in the context of virtual memory management?
Answer: Page replacement is the process of swapping out a page from the main memory to the disk when the page is not currently in use. It is necessary in virtual memory management to ensure that the most relevant pages are kept in the faster main memory.

Q2. Explain the Least Recently Used (LRU) page replacement policy. How does it work?
Answer: LRU replaces the page that has not been used for the longest period. It maintains a counter or stack to track the order of page accesses, and the page at the bottom (least recently used) is chosen for replacement.

Q3. What is the difference between the FIFO and LRU page replacement policies?
Answer: FIFO (First-In-First-Out) replaces the oldest page, while LRU replaces the least recently used page. FIFO doesn't consider the recent usage pattern, whereas LRU does.

Q4. How does the Optimal page replacement policy differ from LRU and FIFO?
Answer: Optimal replacement selects the page that will not be used for the longest period in the future. It is an idealized policy but impractical to implement in real systems due to the need for future knowledge.

Q5. Can you describe a scenario where LRU may not perform well as a page replacement policy?
Answer: LRU may not perform well in scenarios with irregular access patterns or when the working set size is large, as maintaining an accurate LRU order can be resource-intensive.

Q6. Explain the concept of the working set in the context of page replacement.
Answer: The working set is the set of pages that a process is actively using during a specific time interval. Page replacement policies aim to keep the working set in the main memory to minimize page faults.

Q7. How are page faults related to page replacement policies, and why are they significant?
Answer: Page faults occur when a requested page is not in the main memory, leading to a page replacement. Page replacement policies aim to minimize page faults to enhance system performance.

Q8. What are the advantages and disadvantages of using LRU as a page replacement policy?
Answer: Advantages: Effective in capturing temporal locality.
Disadvantages: Implementation can be complex and resource-intensive.

VIVA QUESTIONS:

Q1. What is memory management, and why is it necessary in an operating system?
Answer: Memory management involves allocating and deallocating memory to processes in an organized manner. It is necessary to optimize the use of available memory and prevent conflicts between processes.

Q2. Explain the First Fit memory allocation algorithm. How does it work?
Answer: First Fit allocates the first available block of memory that is large enough to accommodate the process. It starts searching from the beginning of the memory and assigns the first block that fits.

Q3. How does the Best Fit memory allocation algorithm differ from First Fit?
Answer: Best Fit selects the smallest available block of memory that is large enough to accommodate the process. It searches for the best-fit block, minimizing fragmentation.

Q4. Explain the Worst Fit memory allocation algorithm. How does it work?
Answer: Worst Fit allocates the largest available block of memory to the process. It searches for the largest block and assigns it to the process, which may lead to less efficient memory utilization.

Q5. What is external fragmentation, and how does it occur in the context of memory management?
Answer: External fragmentation occurs when free memory is scattered in small, non-contiguous blocks. It happens due to the allocation and deallocation of memory, leading to unused spaces between allocated blocks.

Q6. How do the First Fit, Best Fit, and Worst Fit algorithms differ in their impact on external fragmentation?
Answer: First Fit and Worst Fit may contribute to external fragmentation, as they allocate memory without considering the optimal fit. Best Fit aims to minimize fragmentation by selecting the smallest available block.

Q7. Can you describe a scenario where Best Fit may not perform well in terms of memory utilization?
Answer: Best Fit may not perform well when there are many small free blocks scattered across memory. It may lead to fragmentation as it tries to find the smallest fit.

Q8. What are the advantages and disadvantages of using First Fit for memory allocation?
Answer: Advantages: Simple and easy to implement. Disadvantages: May lead to fragmentation, and not always the most efficient fit.

Q9. What are the advantages and disadvantages of using Best Fit for memory allocation?

Answer: Advantages: Tends to minimize fragmentation. Disadvantages: May be slower than other algorithms due to the search for the best fit.

Q10. What are the advantages and disadvantages of using Worst Fit for memory allocation?
Answer: Advantages: Allocates large blocks, reducing the chance of future allocations.
Disadvantages: May lead to significant fragmentation and inefficient memory utilization.

VIVA QUESTIONS:

Q1. What is the Reader/Writer problem, and why is it relevant in concurrent programming?
Answer: The Reader/Writer problem involves multiple processes accessing a shared resource. Readers can access the resource simultaneously, while writers need exclusive access. It's relevant in scenarios where data integrity must be maintained.

Q2. Explain the difference between a Reader and a Writer in the context of the Reader/Writer problem.
Answer: Readers only read the shared resource and don't modify it, while Writers write or modify the resource. Readers can access the resource simultaneously, but Writers need exclusive access.

Q3. What is the role of semaphores in solving the Reader/Writer problem?
Answer: Semaphores are used to control access to the shared resource. They help ensure exclusive access for Writers and allow multiple Readers to access the resource concurrently.

Q4. How does the solution to the Reader/Writer problem using semaphores ensure data integrity?
Answer: The solution ensures that only one Writer can access the shared resource at a time, preventing conflicts that could lead to data corruption. Readers can access the resource concurrently without any issues.

Q5. Explain the concept of mutual exclusion in the context of the Reader/Writer problem.
Answer: Mutual exclusion ensures that only one process (either a Reader or a Writer) can access the critical section (shared resource) at a time. This prevents data corruption due to simultaneous access.

Q6. What is the purpose of the mutex semaphore in the Reader/Writer problem solution?
Answer: The mutex semaphore is used to achieve mutual exclusion. It ensures that only one process can enter the critical section at a time, preventing conflicts during read or write operations.

Q7. How does the Reader/Writer problem solution handle priority between Readers and Writers?
Answer: The solution typically allows multiple Readers to access the resource simultaneously, giving priority to Readers. However, it ensures that Writers have exclusive access when they need to write to the resource.

Q8. Can you describe a scenario where the absence of synchronization mechanisms could lead to data corruption in the Reader/Writer problem?
Answer: Without synchronization, multiple Writers or a Writer and a Reader could access the shared resource simultaneously, leading to data corruption if they modify the resource concurrently.

Q9. What are the advantages of using semaphores for solving synchronization problems like the Reader/Writer problem?
Answer: Semaphores provide a flexible and efficient way to control access to shared resources. They allow for the implementation of synchronization mechanisms to prevent data corruption and ensure data integrity.

VIVA QUESTIONS:

Q1. What is the Producer-Consumer problem, and why is it relevant in concurrent programming?

Answer: The Producer-Consumer problem involves two types of processes, Producers, and Consumers, accessing a shared, finite-size buffer. Producers add items to the buffer, and Consumers remove items. It's relevant in scenarios where data must be produced and consumed concurrently.

Q2. Explain the roles of Producers and Consumers in the Producer-Consumer problem.

Answer: Producers generate items and add them to a shared buffer, while Consumers retrieve items from the buffer and process them. The goal is to ensure proper synchronization to prevent issues like buffer overflows or underflows.

Q3. How do semaphores help in solving the synchronization issues in the Producer-Consumer problem?

Answer: Semaphores help control access to the shared buffer, ensuring that Producers and Consumers don't access it simultaneously. They prevent issues like race conditions and buffer overflows or underflows.

Q4. How does the solution to the Producer-Consumer problem using semaphores ensure data integrity?

Answer: The solution ensures that Producers and Consumers access the shared buffer in a coordinated manner, preventing issues like data corruption, buffer overflows, or underflows. Semaphores help enforce synchronization.

Q5. How does the Producer-Consumer problem solution handle scenarios where the buffer is full or empty?

Answer: When the buffer is full, Producers are blocked until space is available (Full semaphore). When the buffer is empty, Consumers are blocked until items are available (Empty semaphore), ensuring proper synchronization.

Q6. Can you describe a scenario where the absence of synchronization mechanisms could lead to issues in the Producer-Consumer problem?

Answer: Without synchronization, Producers and Consumers may access the shared buffer simultaneously, leading to issues like race conditions, buffer overflows, or underflows, and data corruption.

Q7. What are the advantages of using semaphores for solving synchronization problems like the Producer-Consumer problem?

Answer: Semaphores provide a structured way to control access to shared resources, preventing race conditions and ensuring synchronization. They are versatile and can be adapted to various synchronization scenarios.

VIVA QUESTIONS:

Q1. What is deadlock, and why is it a concern in operating systems?
Answer: Deadlock is a state where a set of processes is blocked because each process is holding a resource and waiting for another resource acquired by some other process. It's a concern as it can lead to a system's unresponsiveness.

Q2. Explain the Banker's algorithm. What is its primary goal?
Answer: The Banker's algorithm is a deadlock avoidance algorithm that allocates resources to processes in a safe manner, ensuring that the system will not enter into a deadlock state.

Q3. What are the essential components of the Banker's algorithm?
Answer: The Banker's algorithm has matrices for maximum resource allocation, current resource allocation, and available resources. It also maintains a list of processes.

Q4. What is the significance of the "claim edge" in the Banker's algorithm?
Answer: The claim edge represents the maximum resources a process may request during its execution. It is used to determine if a request can be granted without leading the system into an unsafe state.

Q5. Explain the concept of the "safety sequence" in the Banker's algorithm.
Answer: A safety sequence is a sequence of processes that can be executed to completion without entering a deadlock state. If a safety sequence exists, the system is considered safe, and resource allocation is permitted.

Q6. How does the Banker's algorithm assess whether a resource request can be granted safely?
Answer: The algorithm checks if granting the resource request will lead to a safe state by simulating the resource allocation. If the request can be granted without causing an unsafe state, it is allowed.

Q7. What is the role of the "available" vector in the Banker's algorithm?
Answer: The "available" vector represents the available resources in the system. It is used to check if a process's resource request can be granted without causing a deadlock.

Q8. Can you describe a scenario where the Banker's algorithm may unnecessarily restrict resource allocation?
Answer: The Banker's algorithm may restrict resource allocation when it is too conservative in evaluating the safety of resource requests. This conservatism may limit system performance.

Q9. What are the disadvantages or limitations of the Banker's algorithm?
Answer: Banker's algorithm requires knowledge of maximum resource needs, and it may be impractical in situations where this information is not available. It can also be conservative, limiting system utilization.

VIVA QUESTIONS:

Q1. What is file organization, and why is it important in computer systems?
Answer: File organization refers to the way data is stored in a file. It is important as it influences the efficiency of data retrieval, insertion, and deletion operations.

Q2. Explain Indexed File Organization. How does it work, and what is the role of the index?
Answer: In Indexed File Organization, an index is maintained separately from the actual data. The index contains key-value pairs, where the key is used for quick data retrieval. It enhances search and retrieval efficiency.

Q3. What are the advantages and disadvantages of Indexed File Organization?
Answer:
Advantages include faster data retrieval, efficient searching.
Disadvantages may include increased storage requirements for the index.

Q4. Explain Heap File Organization. How is data stored in a heap file?
Answer: In Heap File Organization, data is stored in an unstructured manner without any specific order. New data is appended at the end of the file. It is simple but may lead to slower search operations.

Q5. What are the advantages and disadvantages of Heap File Organization?
Answer: Advantages include simplicity and ease of insertion. Disadvantages may include slower search operations due to lack of structure.

Q6. How does hashing contribute to efficient data retrieval?
Answer: Hashing contributes to efficient data retrieval by providing a direct and quick way to locate and access data based on its key. The hash function calculates an index or address based on the key, and this index directly points to the location where the corresponding data is stored.

Q7. What are the advantages and disadvantages of Hash File Organization?
Answer: Advantages include quick and direct access to data. Disadvantages may include potential collisions and the need for an efficient hash function.

Q8. Explain Sequential File Organization. How is data stored in a sequential file?
Answer: In Sequential File Organization, data is stored in a linear order, typically sorted by a specific key. New data is added at the end, and efficient sequential access is supported.

Q9. What are the advantages and disadvantages of Sequential File Organization?
Answer:
Advantages include efficient sequential access.
Disadvantages may include slower random access and the need for sorting after each insertion.