Name-Abhi Mehta
Reg No.-221080001

Experiment-09

Aim-To implement all file allocation strategies:
1.Contiguous Allocation.
2.Linked Allocation.
3.Indexed Allocation.

Theory:
File allocation refers to the process of mapping logical file records to physical disk blocks. When a file is created or modified, the operating system needs to determine where on the disk the file's data should be stored. The primary goal of file allocation is to provide efficient storage and retrieval of files while minimizing disk fragmentation and overhead.
The fundamental concepts and considerations in file allocation theory include:

Disk Space Management:

- The operating system maintains a free space list or a free space management data structure to keep track of available disk blocks.
- When a file is created or extended, the file allocation strategy selects free blocks from this list and allocates them to the file.
- When a file is deleted or truncated, the previously allocated blocks are returned to the free space list for future reuse.

Fragmentation:

- Fragmentation occurs when files are not stored in contiguous disk blocks, leading to the scattering of file blocks across the disk.
- External fragmentation refers to the situation where free blocks are scattered, making it difficult to allocate contiguous blocks for new or growing files.
- Internal fragmentation occurs when a file's last block is partially filled, leading to wasted disk space within that block.
- Minimizing fragmentation is a key consideration in file allocation theory to ensure efficient storage and retrieval.

Access Patterns:

- Different file allocation strategies are optimized for different access patterns, such as sequential access (reading or writing a file from start to finish) or random access (accessing arbitrary portions of a file).
- Sequential access benefits from contiguous block allocation, while random access may be more efficient with linked or indexed allocation strategies.

File Growth and Shrinkage:

- File allocation strategies should handle dynamic file growth and shrinkage efficiently, without requiring excessive data movement or disk head movement.
- Strategies like linked allocation and indexed allocation can accommodate file growth more easily than contiguous allocation.

Metadata Management:

- File allocation strategies require metadata (information about the file's structure and block locations) to be stored and managed efficiently.
- Directory entries, file allocation tables, or index blocks are used to store this metadata, introducing overhead and potential performance trade-offs.

Performance Considerations:

- File allocation strategies aim to minimize disk seeks (movements of the disk head) and optimize data transfer rates.
- Strategies that promote contiguous block allocation or efficient block access patterns can improve overall disk performance.

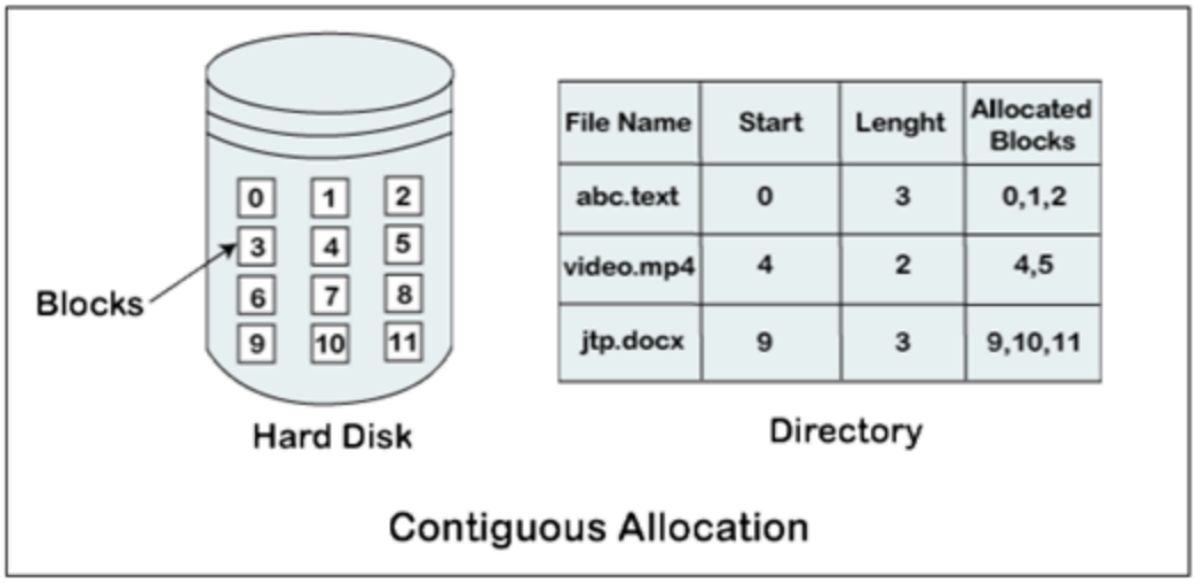
Reliability and Recovery:

- File allocation strategies should support reliable data storage and enable efficient recovery in case of system failures or disk errors.
- Techniques like journaling or redundancy can be employed to enhance reliability and recoverability.

File allocation theory explores these concepts and trade-offs, aiming to develop efficient and robust strategies that meet the diverse requirements of modern file systems and storage technologies.

File Allocation Strategies:

File allocation strategies are techniques used by operating systems to manage how files are stored on disk. Different allocation strategies offer trade-offs in terms of efficiency, performance, and storage utilization.Following are the three main file allocation strategies you mentioned:



Contiguous Allocation:

- In this strategy, each file is stored as a contiguous block of consecutive disk blocks.
- The directory entry for the file contains the starting block number and the length (in blocks) of the file.

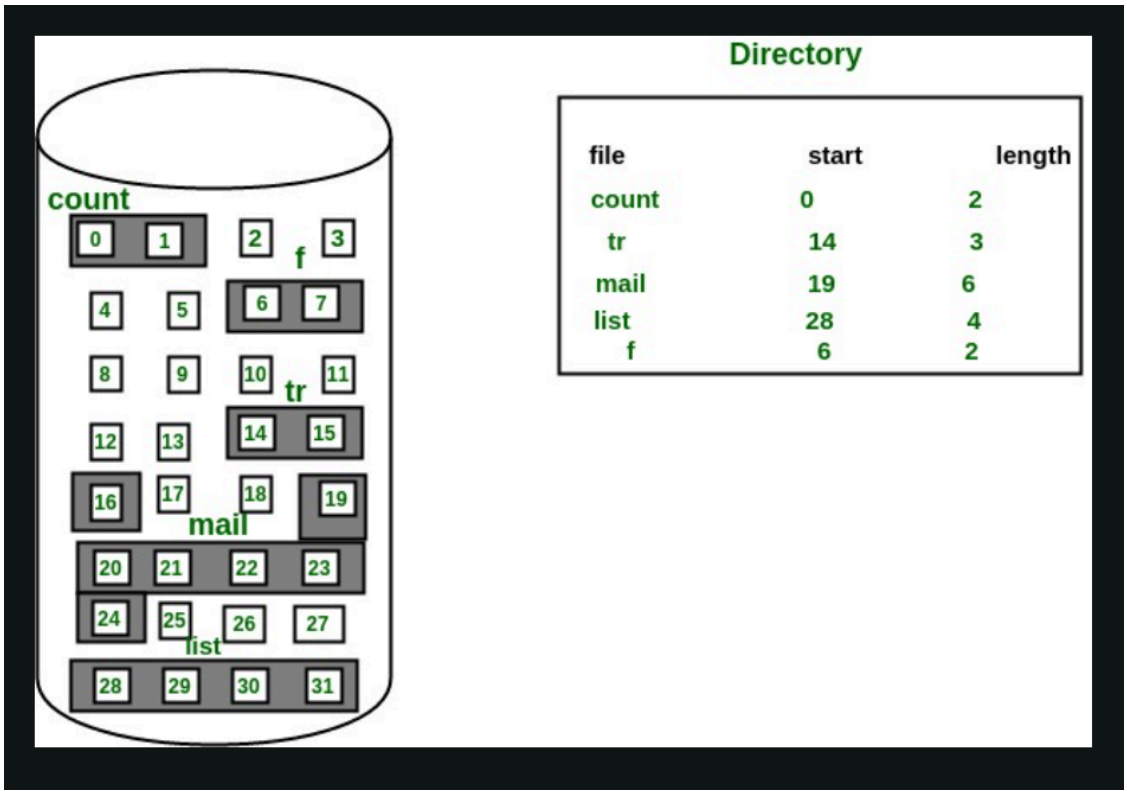
Advantages:

- Simplicity: Easy to implement and understand.
- Efficient for sequential access (reading/writing the file from start to finish).

Disadvantages:

- External fragmentation: Over time, disk space becomes fragmented, and it may be difficult to find a contiguous block of free space large enough to store a new file or extend an existing one.

- Inefficient for random access and file growth: If a file needs to grow, it may need to be moved to a different location on the disk, which can be time-consuming.



Linked Allocation:

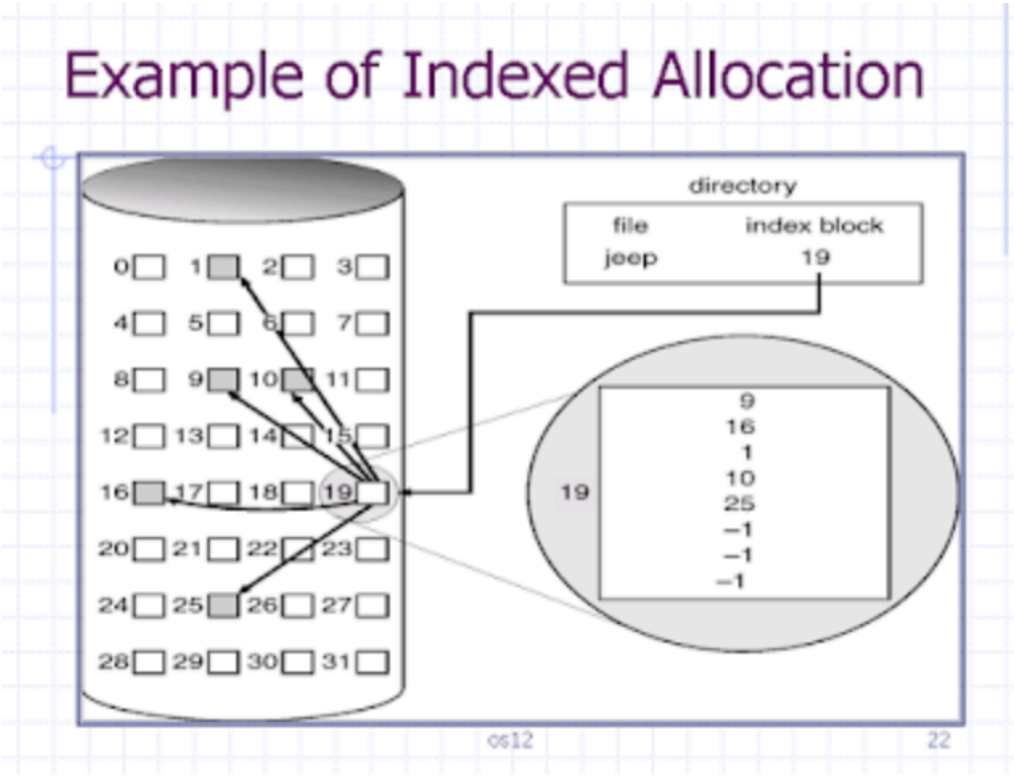
- In this strategy, each file is stored as a linked list of disk blocks.
- The directory entry for the file contains the block number of the first block in the linked list.
- Each block in the linked list contains a pointer to the next block in the sequence, except for the last block, which has a null pointer.

Advantages:

- No external fragmentation: Files can be stored in any available disk blocks, making efficient use of disk space.
- Efficient for random access and file growth: Files can grow dynamically by allocating new blocks and updating the linked list.

Disadvantages:

- Overhead: Each block requires extra space to store the pointer to the next block, reducing the effective storage capacity.
- No sequential access: Reading or writing a file sequentially requires following the linked list, which can be less efficient than contiguous access.



Indexed Allocation:

- In this strategy, a single index block is used to store the block numbers of all the blocks that make up the file.
- The directory entry for the file contains the block number of the index block.
- The index block contains an array of block numbers that point to the actual data blocks of the file.

  Advantages:
- Support for large files: An index block can store a large number of block pointers, allowing for efficient handling of large files.

- Efficient for both sequential and random access: Sequential access is achieved by reading the index block and then the data blocks in order, while random access is possible by accessing the desired block directly.

  Disadvantages:
- Overhead: The index block adds extra overhead, especially for small files where the index block may be larger than the file itself.
- Potential fragmentation: If the file grows beyond the capacity of the index block, additional index blocks may be required, leading to potential fragmentation.

These file allocation strategies have their own strengths and weaknesses, and the choice of which one to use depends on factors such as the expected file sizes, access patterns, and performance requirements of the system. In practice, modern file systems often combine or extend these basic strategies to improve performance and flexibility.

Implementation:

Contiguous:

```c
#include <stdio.h>
#include <stdbool.h>
#include <time.h>
int main()
{
int i, j, k, tot_blocks, n_blocks, n_files, sb[50], l[50];
clock_t start, end;
start = clock();
//Enter the total number of blocks
printf("\n Enter the total number of blocks : ");
scanf("%d", &tot_blocks);
int memory[tot_blocks];
for (i = 0; i < tot_blocks; i++)
memory[i] = -1;
printf(" Enter the number of files : ");
scanf("%d", &n_files);
bool file_alloc[n_files];
int alloc_info[n_files];
printf("\n");
for (i = 0; i < n_files; i++)
{
printf(" Enter the starting block number of file %d : ", i + 1);
scanf("%d", &sb[i]);
printf(" Enter the length of file %d : ", i + 1);
scanf("%d", &l[i]);
sb[i]--;
```

```
file_alloc[i] = false;
alloc_info[i] = -1;
}
for (i = 0; i < n_files; i++)
{
j = sb[i];
alloc_info[i] = j;
file_alloc[i] = true;
if (memory[j] != -1)
{
printf("\n Cannot allocate file %d.", i + 1);
alloc_info[i] = -1;
file_alloc[i] = false;
continue;
}
else
{
for (j = sb[i]; j < sb[i] + l[i]; j++)
{
if (memory[j] != -1)
{
printf("\n Not enough space to allocate file %d.", i + 1);
file_alloc[i] = false;
for(k = j-1; k >= sb[i]; k--)
{
memory[k] = -1;
}
alloc_info[i] = -1;
break;
}
memory[j] = i + 1;
}
}
}
printf("\n\n File allocation information : ");
printf("\n File no. Starting block no. Length \n");
for (i = 0; i < n_files; i++)
{
if (file_alloc[i] == true)
{
printf(" %d" "%d" "%d \n", i + 1, alloc_info[i] + 1, l[i]);
}
else if (file_alloc[i] == false)
{
printf(" %d Not allocated %d \n", i + 1, l[i]);
}
}
printf("\n");
for (i = 0; i < tot_blocks; i++)
printf(" %d ", memory[i]);
printf("\n");
end = clock();
double time_required = ((double) (end - start)) / CLOCKS_PER_SEC;
printf("Time required: %lf\n", time_required);
return 0;
}
```

Linked Allocation:

os8_linked.c > main()

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
int tot_blocks, n_files, empty_blocks;

//Function to initialise block
int initialize(bool blocks[], int next[], int SB[], int EB[])
{
for (int i = 0; i < tot_blocks; i++)
{
blocks[i] = false; // 0 - block free
next[i] = -1;
```

```
// gives next bl_no
}
for (int i = 0; i < n_files; i++)
{
SB[i] = -1; // Starting Block_nos
EB[i] = -1; //Ending Block_nos
}
return 0;
}

//Function to get block number
int getBN(bool blocks[])
{
int j = 0, val = -1;
srand(time(0));
while (j == 0 && empty_blocks > 0)
{
val = rand() % tot_blocks;
if (blocks[val] == false)
{
blocks[val] = true;
j = 1;
}
}
return val;
}

//Main function
int main()
{
clock_t start, end;
start = clock();
int i, j;
printf("\n Enter the total number of blocks : ");
scanf("%d", &tot_blocks);
printf(" Enter total number of files : ");
scanf("%d", &n_files);
bool blocks[tot_blocks];
int next[tot_blocks];
int SB[n_files];
int EB[n_files];
int fileLen[n_files];
int val, prev;
empty_blocks = tot_blocks;
initialize(blocks, next, SB, EB);
printf("\n Enter the length of the each file : \n");
for (i = 0; i < n_files; i++)
{
printf(" File %d - ", i);
scanf("%d", &fileLen[i]);
}
for (i = 0; i < n_files; i++)
{
if (fileLen[i] <= empty_blocks)
{
```

```
j = 1;
SB[i] = getBN(blocks);
j++;
prev = SB[i];
while (j <= fileLen[i])
{
next[prev] = getBN(blocks);
prev = next[prev];
j++;
```

```
78  empty_blocks--;
79  }
80  EB[i] = prev;
81  }
82  }
83  printf("\n File no. Starting block Ending Block\n");
84  for (i = 0; i < n_files; i++)
85  {
86  if (SB[i] != -1)
87  printf(" %d%d%d\n", i, SB[i],EB[i]);
88  else
89  printf(" %d Couldn't be allocated\n", i);
90  }
91  printf("\n File no.\t Path\n");
92  for (i = 0; i < n_files; i++)
93  {
94  printf(" %d\t ", i);
95  if (SB[i] != -1)
96  {
97  int p = SB[i];
98  while (p != -1)
99  {
100 printf(" %d -> ", p);
101 p = next[p];
102 }
103 printf(" NULL\n");
104 }
105 else
106 printf(" None");
107 }
108 printf("\n");
109 end = clock();
110 printf("Time taken: %f\n", (double)(end - start) / CLOCKS_PER_SEC);
111 return 0;
112 }
```

Indexed Allocation:

C os8_index.c > main()

```
1   #include<stdio.h>
2   #include<stdlib.h>
3   #include<time.h>
4   int main()
5   {
6   clock_t start, end;
7   double cpu_time_used;
8   start = clock();
9   int f[100], index[100],i, n, st, len, j, c, k, ind,count=0;
10  for(i=0;i<100;i++)
11  f[i]=0;
12  x:printf("Enter the index block: ");
13  scanf("%d",&ind);
14  if(f[ind]!=1)
15  {
16  printf("Enter no of blocks needed and blocks for the index %d on the disk : \n", ind);
17  scanf("%d",&n);
18  }
19  else
20  {
21  printf("%d index is already allocated \n",ind);
```

```c
goto x;
}
y:
count = 0;
for (i = 0; i < n; i++)
{
scanf("%d", &index[i]);
if (f[index[i]] == 1)
count++;
}
if(count==0)
{
for(j=0;j<n;j++)
f[index[j]]=1;
printf("Allocated\n");
printf("File Indexed\n");
for(k=0;k<n;k++)
printf("%d-------->%d : %d\n",ind,index[k],f[index[k]]);
}
else
{
printf("File in the index is already allocated \n");
printf("Enter another file indexed");
goto y;
}
printf("Do you want to enter more file(Yes - 1/No - 0)");
scanf("%d", &c);
if(c==1)
goto x;
else
end = clock();
cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
printf("Time taken: %f\n", cpu_time_used);
exit(0);
return 0;
}
```

Outputs:

Contiguous:

```
Enter the total number of blocks : 16
Enter the number of files : 3

Enter the starting block number of file 1 : 1
Enter the length of file 1 : 6
Enter the starting block number of file 2 : 8
Enter the length of file 2 : 3
Enter the starting block number of file 3 : 13
Enter the length of file 3 : 2


File allocation information :
File no. Starting block no. Length
116
283
3132

 1  1  1  1  1  1  -1  2  2  2  -1  -1  3  3  -1  -1
Time required: 0.000442
```

```
Enter the total number of blocks : 12
Enter the number of files : 3

Enter the starting block number of file 1 : 1
Enter the length of file 1 : 6
Enter the starting block number of file 2 : 3
Enter the length of file 2 : 8
Enter the starting block number of file 3 : 7
Enter the length of file 3 : 10

Cannot allocate file 2.
Not enough space to allocate file 3.

File allocation information :
File no. Starting block no. Length
116
2 Not allocated 8
3 Not allocated 10

1  1  1  1  1  1  -1  -1  -1  -1  -1  -1
Time required: 0.000563
```

Linked:

```
Enter the total number of blocks : 10
Enter total number of files : 3

Enter the length of the each file :
File 0 - 3
File 1 - 4
File 2 - 3

File no. Starting block Ending Block
024
109
258

File no.        Path
0       2 ->  3 ->  4 ->  NULL
1       0 ->  6 ->  1 ->  9 ->  NULL
2       5 ->  7 ->  8 ->  NULL

Time taken: 0.000353
```

Indexed:

```
Enter the index block: 10
Enter no of blocks needed and blocks for the index 10 on the disk :
6
2
3
4
8
7
6
Allocated
File Indexed
10-------->2 : 1
10-------->3 : 1
10-------->4 : 1
10-------->8 : 1
10-------->7 : 1
10-------->6 : 1
Do you want to enter more file(Yes - 1/No - 0)1
Enter the index block: 9
Enter no of blocks needed and blocks for the index 9 on the disk :
2
9
8
File in the index is already allocated
Enter another file indexed5
9
Allocated
File Indexed
9-------->5 : 1
9-------->9 : 1
Do you want to enter more file(Yes - 1/No - 0)0
Time taken: 0.000819
```

Analysis:

Based on the time efficiency figures , it is clear that contiguous allocation is the fastest (0.000442 seconds), followed by linked allocation (0.000563 seconds), and indexed allocation is the slowest (0.000819 seconds).

Conclusion:

In terms of time efficiency, contiguous allocation is the fastest among the three strategies. Linked allocation comes second, and indexed allocation is the slowest. The additional overhead involved in accessing and managing index blocks contributes to the higher time requirement for indexed allocation.

Time Efficiency:

- Contiguous allocation: 0.000442 seconds (fastest)
- Linked allocation: 0.000563 seconds
- Indexed allocation: 0.000819 seconds (slowest)

Based on these figures, contiguous allocation is approximately 1.27 times faster than linked allocation and 1.85 times faster than indexed allocation for file access operations.

Memory Efficiency:

- Contiguous allocation has less memory overhead but may suffer from external fragmentation.
- Linked allocation requires more memory overhead due to the storage of pointers for each block, but it can handle dynamic file sizes efficiently.
- Indexed allocation requires additional memory for index blocks, resulting in higher memory overhead. However, it allows convenient access to files.

Fragmentation:

- Contiguous allocation leads more easily to external fragmentation, especially when files are frequently created, modified, and deleted.
- Linked allocation and indexed allocation result in minimal fragmentation as blocks are allocated on demand, similar to each other in this respect.

Access Efficiency:

- Contiguous allocation provides efficient sequential access but may be less efficient for random access and direct block operations.
- Linked allocation provides efficient sequential access and tolerable direct block access due to the traversal of pointers.
- Indexed allocation provides good direct block access but may have higher overhead for sequential access and random access due to the need for index block searching.

Based on this analysis, the choice of file allocation strategy should consider the specific requirements and trade-offs involved. Contiguous allocation may be preferred for systems with fixed-size files and primarily sequential access patterns, where time efficiency is crucial, and memory overhead is not a significant concern. Linked allocation can be a good choice for systems with dynamic file sizes and a balance between sequential and random access patterns. Indexed allocation may be suitable for systems that require efficient direct block access and can accommodate the additional memory overhead for index blocks.

—----------**END**—----------