Week 12

1. **WAP to demonstrate the contiguous file allocation techniques.**

**Ans.**

**AIM:** A program to demonstrate the contiguous file allocation techniques.

**PROCEDURE:**

In this allocation strategy, each file occupies a set of contiguous blocks on the disk. This strategy is best suited. For sequential files, the file allocation table consists of a single entry for each file. It shows the filenames, starting block of the file and size of the file. The main problem with this strategy is, it is difficult to find the contiguous free blocks in the disk and some free blocks could happen between two files.

**Algorithm for Sequential File Allocation:**

Step 1: Start the program.

Step 2: Get the number of memory partition and their sizes.

Step 3: Get the number of processes and values of block size for each process.

Step 4: First fit algorithm searches all the entire memory block until a hole which is big enough is encountered. It allocates that memory block for the requesting process.

Step 5: Best-fit algorithm searches the memory blocks for the smallest hole which can be allocated to requesting process and allocates it.

Step 6: Worst fit algorithm searches the memory blocks for the largest hole and allocates it to the process.

Step 7: Analyses all the three memory management techniques and display the best algorithm which utilizes the memory resources effectively and efficiently.

Step 8: Stop the program.

**CODE:**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_BLOCKS 50

int main() {

int f[MAX\_BLOCKS] = {0}; // Initialize array to 0

int st, len, c;

while (1) {

printf("Enter starting block and length of file: ");

if (scanf("%d %d", &st, &len) != 2) {

printf("Invalid input. Exiting...\n");

exit(EXIT\_FAILURE);

}

if (st < 0 || st >= MAX\_BLOCKS || len <= 0 || st + len > MAX\_BLOCKS) {

printf("Invalid range. Exiting...\n");

exit(EXIT\_FAILURE);

}

int count = 0;

for (int k = st; k < st + len; k++) {

if (f[k] == 0) {

count++;

}

}

if (count == len) {

for (int j = st; j < st + len; j++) {

f[j] = 1;

printf("Block %d allocated\n", j);

}

} else {

printf("Not enough consecutive free blocks to allocate the file.\n");

}

printf("Do you want to enter more files? (1 for Yes, 0 for No): ");

if (scanf("%d", &c) != 1 || (c != 0 && c != 1)) {

printf("Invalid input. Exiting...\n");

exit(EXIT\_FAILURE);

}

if (c == 0) {

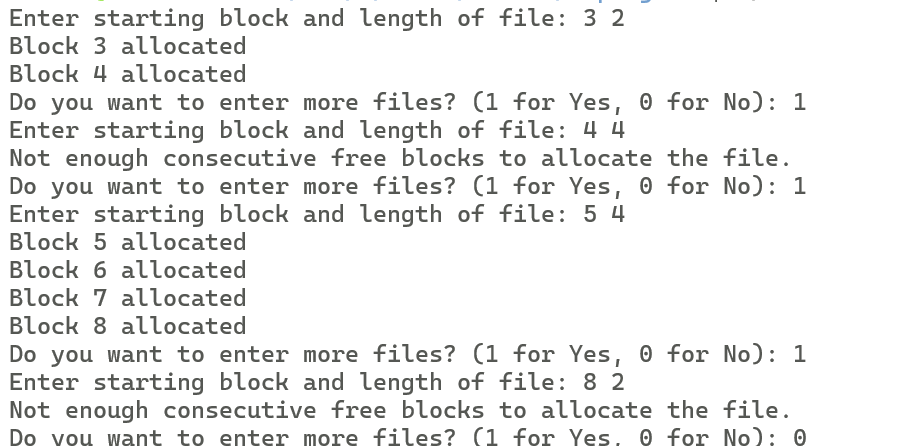
break;

}

}

return 0;

}



1. **WAP to demonstrate the Linked File allocation techniques**

**Ans.**

**AIM:** A program to demonstrate the Linked File allocation techniques

**PROCEDURE:**

It is easy to allocate the files because allocation is on an individual block basis. Each block contains a pointer to the next free block in the chain. Here also the file allocation table consisting of a single entry for each file. Using this strategy any free block can be added to a chain very easily. There is a link between one block to another block, that’s why it is said to be linked allocation. We can avoid the external fragmentation.

**Algorithm for Linked File Allocation:**

Step 1: Create a queue to hold all pages in memory

Step 2: When the page is required replace the page at the head of the queue

Step 3: Now the new page is inserted at the tail of the queue

Step 4: Create a stack

Step 5: When the page fault occurs replace page present at the bottom of the stack

Step 6: Stop the allocation.

**CODE:**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_BLOCKS 50

int main() {

int f[MAX\_BLOCKS] = {0}; // Initialize array to 0

int p, st, len, c;

printf("Enter how many blocks already allocated: ");

if (scanf("%d", &p) != 1 || p < 0 || p > MAX\_BLOCKS) {

printf("Invalid input. Exiting...\n");

exit(EXIT\_FAILURE);

}

printf("Enter blocks already allocated: ");

for (int i = 0; i < p; i++) {

int a;

if (scanf("%d", &a) != 1 || a < 0 || a >= MAX\_BLOCKS) {

printf("Invalid input. Exiting...\n");

exit(EXIT\_FAILURE);

}

f[a] = 1;

}

while (1) {

printf("Enter index of starting block and length: ");

if (scanf("%d %d", &st, &len) != 2 || st < 0 || st >= MAX\_BLOCKS || len <= 0) {

printf("Invalid input. Exiting...\n");

exit(EXIT\_FAILURE);

}

int k = len;

if (f[st] == 0) {

for (int j = st; j < st + k; j++) {

if (j >= MAX\_BLOCKS || f[j] == 1) {

printf("%d Block is already allocated \n", j);

k++;

} else {

f[j] = 1;

printf("%d-------->%d\n", j, f[j]);

}

}

} else {

printf("%d starting block is already allocated \n", st);

}

printf("Do you want to enter more files? (Yes - 1 / No - 0): ");

if (scanf("%d", &c) != 1 || (c != 0 && c != 1)) {

printf("Invalid input. Exiting...\n");

exit(EXIT\_FAILURE);

}

if (c == 0) {

break;

}

}

return 0;

}

**OUTPUT:**

**A screenshot of a computer

Description automatically generated**

**3.WAP to demonstrate Indexed file allocation techniques**

**Ans.**

**AIM:** A program to demonstrate Indexed file allocation techniques

**PROCEDURE:**

Indexed allocation supports both sequential and direct access files. The file indexes are not physically stored as a part of the file allocation table. Whenever the file size increases, we can easily add some more blocks to the index. In this strategy, the file allocation table contains a single entry for each file. The entry consisting of one index block, the index blocks having the pointers to the other blocks. No external fragmentation.

**Algorithm for Indexed File Allocation:**

Step 1: Initialize an array index of size N to store file indices. Initialize another array allocation of size N to store block allocations.

Step 2:Initialize both arrays such that all elements are set to -1, indicating that no file is currently indexed or allocated.

Step3:File Indexing:When a new file is created, allocate an index block for it.

Step4: Search for a free entry in the index array. If found, store the index block number in that entry.

Step5: If no free entry is found, report that the index table is full and the file cannot be created.

Step6:File Allocation:When a file needs to be stored on the disk, allocate data blocks for it.Find a free block on the disk for each block required by the file. Store the block numbers in the data portion of the index block allocated to the file.

Step7:Reading and Writing:To read a file, first read its index block to determine the location of its data blocks.

To write to a file, if additional blocks are required, allocate new blocks and update the index block accordingly.

Keep track of the file size to know how many blocks are allocated to it.

Step 8:Deletion:When a file is deleted, release its allocated data blocks.

Also, release its index block from the index array.

Step 9: Error Handling:Handle errors such as insufficient disk space, full index table, or invalid file operations.

**CODE:**

#include <stdio.h>

#include <stdlib.h>

#define MAX\_BLOCKS 50

int main() {

int f[MAX\_BLOCKS] = {0}; // Initialize array to 0

int index[MAX\_BLOCKS], n, ind, c;

while (1) {

printf("Enter the index block: ");

scanf("%d", &ind);

if (ind < 0 || ind >= MAX\_BLOCKS) {

printf("Invalid index block. Exiting...\n");

exit(EXIT\_FAILURE);

}

if (f[ind] != 1) {

printf("Enter the number of blocks needed for index %d on the disk: \n", ind);

scanf("%d", &n);

if (n < 1 || n > MAX\_BLOCKS) {

printf("Invalid number of blocks. Exiting...\n");

exit(EXIT\_FAILURE);

}

printf("Enter block numbers:\n");

for (int i = 0; i < n; i++) {

scanf("%d", &index[i]);

if (index[i] < 0 || index[i] >= MAX\_BLOCKS) {

printf("Invalid block number. Exiting...\n");

exit(EXIT\_FAILURE);

}

if (f[index[i]] == 1) {

printf("Block %d is already allocated. Exiting...\n", index[i]);

exit(EXIT\_FAILURE);

}

}

printf("Allocated\n");

printf("File Indexed\n");

for (int j = 0; j < n; j++) {

f[index[j]] = 1;

printf("%d-------->%d : %d\n", ind, index[j], f[index[j]]);

}

} else {

printf("%d index is already allocated. Enter another index.\n", ind);

continue;

}

printf("Do you want to enter more files? (Yes - 1 / No - 0): ");

scanf("%d", &c);

if (c == 0) {

break;

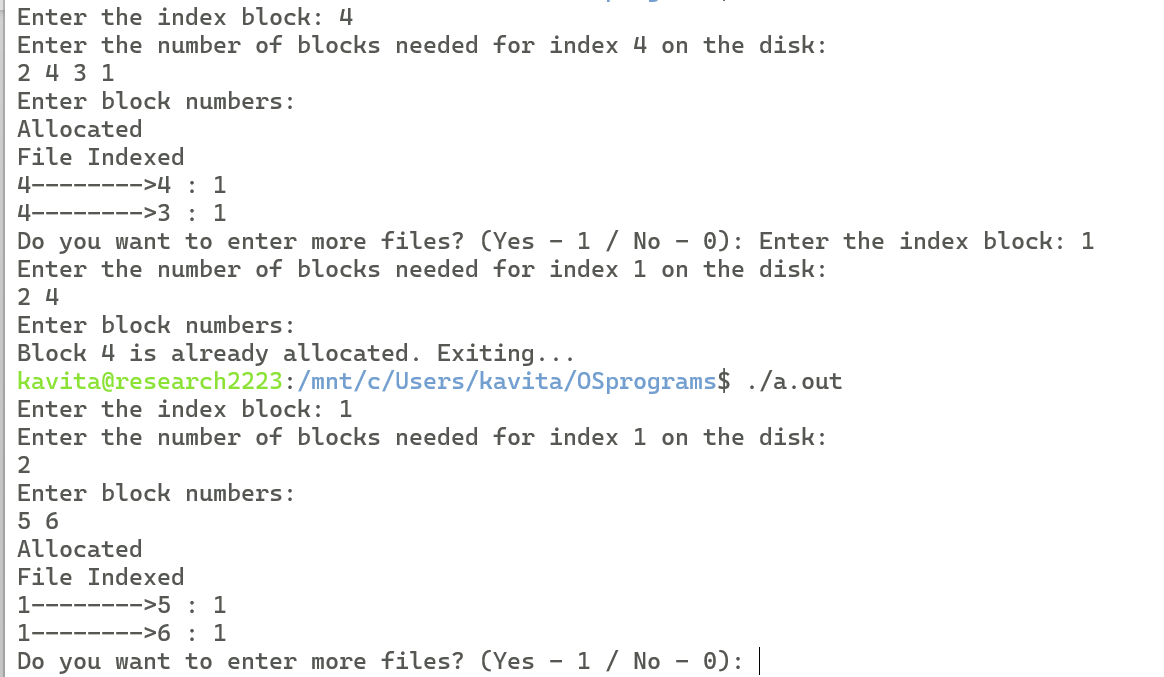
}

}

return 0;

}

OUTPUT:

****

Q)Demonstrate the usage of getuid, setuid, brk, nice and sleep.

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/time.h>

int main() {

// getuid and setuid

uid\_t uid = getuid();

printf("Current user ID: %d\n", uid);

if (setuid(0) == -1) {

perror("setuid");

exit(EXIT\_FAILURE);

}

uid = getuid();

printf("New user ID: %d\n", uid);

// brk

void \*original\_brk = sbrk(0);

printf("Original break value: %p\n", original\_brk);

int increment = 4096; // Increment by 4KB

void \*new\_brk = sbrk(increment);

if (new\_brk == (void \*)-1) {

perror("sbrk");

exit(EXIT\_FAILURE);

}

printf("New break value: %p\n", new\_brk);

// nice

int current\_priority = nice(0);

printf("Current process priority: %d\n", current\_priority);

int nice\_value = 10; // Increment priority by 10

if (nice(nice\_value) == -1) {

perror("nice");

exit(EXIT\_FAILURE);

}

current\_priority = nice(0);

printf("New process priority: %d\n", current\_priority);

// sleep

printf("Sleeping for 3 seconds...\n");

sleep(3);

printf("Awake!\n");

return 0;

}