**LAB 1**

**Aim:**

To implement Fixed Size Memory Allocation

**Theory:**

In case of fixed size partitioning, the block sizes available for process allocation are fixed by the operating system (and thus cannot be allocated to fit the processes exactly). In lieu of this, allocation is done based on 3 possible schemes:

1.Best fit: Where a process is allocated to the smallest block which is greater than its size.

2.Worst fit: Where a process is allocated to the largest available.

3.First fit: Where a process is allocated to the first free block encountered (which is big enough to hold) found during scan of the main memory**.**

**METHOD:**

Implementation has been done with C, using arrays to simulate the memory blocks available (whose sizes have been hard coded into the system). Number of processes to allocate and their sizes are taken through user input, and allocation status after first, best and worst fit allocation is printed to the console.

a. Best Fit

Allocate the process to the partition which is first smallest sufficient partition among the free available partition.

#include<stdio.h> // Include library

int main()

{

int fragments, process; // **initialise variables for fragments and processes**

printf("Enter number of fragments:");

scanf("%d", &fragments);

printf("Enter number of processes:");

scanf("%d", &process);

int frag[fragments], proc[process], manage[process]; // **array for fragments and process**

printf("Enter size of fragments:");

for(int i = 0; i < fragments; i++)

scanf("%d", &frag[i]); // **take input for size of each fragment**

printf("Enter size of processes:");

for(int i = 0; i < process; i++)

{

scanf("%d", &proc[i]); // **take input for size of processes**

manage[i] = -1;

}

for(int i = 0; i < process; i++)

{

int lowest;

int pos = -1, first = 0;

for(int j = 0; j < fragments; j++)

{

if(proc[i] <= frag[j]) // **find the best fit for process i**

{

if(first == 0) // **make first one as best fit**

{

first = 1;

lowest = frag[j];

pos = j;

}

else

{

if(lowest > frag[j]) // **find better fit than first one.**

{

lowest = frag[j];

pos = j;

}

}

}

}

if(pos != -1) // fragment allocated

{

frag[pos] -= proc[i]; // decrement size of fragment

printf("Process %d allocated to fragment %d\n", i + 1, pos + 1);

manage[i] = pos;

}

}

printf("Do you want to delete a process? Press Y or N");

char s;

scanf("%c", &s);

if(s == 'Y')

{

printf("Enter number of processes you want to delete");

int n;

scanf("%d", &n);

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

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

{

int num;

scanf("%d", &num);

int pos = manage[num];

manage[num - 1] = -1; // **process num is deleted**

frag[pos] += proc[num – 1]; // **increment size of fragment**

}

}

for(int i = 0; i < fragments; i++)

{

printf("Fragment %d has %d space available!", i + 1, frag[i]);

}

return 0;

}

b. First Fit

In the first fit, partition is allocated which is first sufficient from the top of Main Memory.

#include<stdio.h>

int main()

{

int fragments, process; // **initialise variables for fragments and processes**

printf("Enter number of fragments:");

scanf("%d", &fragments);

printf("Enter number of processes:");

scanf("%d", &process);

int frag[fragments], proc[process], manage[process]; // **array for fragments and process**

printf("Enter size of fragments:");

for(int i = 0; i < fragments; i++)

scanf("%d", &frag[i]); // **take input for size of each fragment**

printf("Enter size of processes:");

for(int i = 0; i < process; i++)

{

scanf("%d", &proc[i]); // **take input for size of processes**

manage[i] = -1;

}

for(int i = 0; i < process; i++)

{

int pos = -1;

for(int j = 0; j < fragments; j++)

{

if(proc[i] <= frag[j]) // **find the first fit for process i**

{

pos = j;

frag[pos] -= proc[i];

printf("Process %d allocated to fragment %d\n", i + 1, pos + 1);

manage[i] = pos;

break;

}

}

}

printf("Do you want to delete a process? Press Y or N");

char s;

getchar();

if(s == 'Y')

{

printf("Enter number of processes you want to delete");

int n;

scanf("%d", &n);

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

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

{

int num;

scanf("%d", &num);

int pos = manage[num];

manage[num - 1] = -1; // **process num is deleted**

frag[pos] += proc[num – 1]; // **increment size of fragment**

}

}

for(int i = 0; i < fragments; i++)

{

printf("Fragment %d has %d space available!", i + 1, frag[i]);

}

return 0;

}

c. Worst Fit

Allocate the process to the partition which is largest sufficient among the freely available partitions available in the main memory.

#include<stdio.h>

int main()

{

int fragments, process; // **initialise variables for fragments and processes**

printf("Enter number of fragments:");

scanf("%d", &fragments);

printf("Enter number of processes:");

scanf("%d", &process);

int frag[fragments], proc[process], manage[process]; // **array for fragments and process**

printf("Enter size of fragments:");

for(int i = 0; i < fragments; i++)

scanf("%d", &frag[i]); // **take input for size of each fragment**

printf("Enter size of processes:");

for(int i = 0; i < process; i++)

{

scanf("%d", &proc[i]); // **take input for size of processes**

manage[i] = -1;

}

for(int i = 0; i < process; i++)

{

int highest;

int pos = -1, first = 0;

for(int j = 0; j < fragments; j++)

{

if(proc[i] <= frag[j]) // **find the first fit for process i**

{

if(first == 0)  **// make first one as worst fit**

{

first = 1;

highest = frag[j];

pos = j;

}

else

{

if(highest < frag[j]) // **find worse fit than first one.**

{

highest = frag[j];

pos = j;

}

}

}

}

if(pos != -1)

{

frag[pos] -= proc[i];

printf("Process %d allocated to fragment %d\n", i + 1, pos + 1);

manage[i] = pos;

}

}

printf("Do you want to delete a process? Press Y or N");

char s;

scanf("%c", &s);

if(s == 'Y')

{

printf("Enter number of processes you want to delete");

int n;

scanf("%d", &n);

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

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

{

int num;

scanf("%d", &num);

int pos = manage[num];

manage[num - 1] = -1; // **process num is deleted**

frag[pos] += proc[num – 1]; // **increment size of fragment**

}

}

for(int i = 0; i < fragments; i++)

{

printf("Fragment %d has %d space available!", i + 1, frag[i]);

}

return 0;

}

**AIM:**

To simulate contiguous memory allocation with blocks of variable size, and demonstrate allocation in cases of:

1. Best fit scheme

2. Worst fit scheme

3. First fit scheme

**THEORY:**

In case of variable size blocks, the block sizes available for process allocation are dynamic, and are set to fit processes exactly, and remain fixed once the process is terminated. Similar to cases of fixed block allocation, we have the same three schemes:

1. Best fit: Where a process is allocated to the smallest block which is greater than its size.

2. Worst fit: Where a process is allocated to the largest available.

3. First fit: Where a process is allocated to the first free block encountered (which is big enough to hold) found during scan of the main memory.

**METHOD:**

Implementation has been done with C, using arrays to simulate the memory blocks available (whose sizes are fixed after the initial processes are terminated). Number of processes to allocate and their sizes are taken through user input, and allocation status after first, best and worst fit allocation is printed to the console.

**CODE:**

#include&lt;stdio.h&gt;

#include&lt;stdlib.h&gt;

struct Node {

int from;

int size;

struct Node \*next;

};

struct Node \*freeMemory, \*allocMemory;

// **Inserting the process in the Linked List**

void insertSorted(struct \*node, int from, int size){

if (!node){

struct Node \*tmp = (struct Node\*) mallocMemory(sizeof(struct Node));

tmp-&gt;from = from;

tmp-&gt;size = size;

freeMemory = tmp;

}

else if ((node-&gt;from &lt; from &amp;&amp; node-&gt;next &amp;&amp; node-&gt;next- &gt;from &gt; from) || !node-

&gt;next){

struct Node \*tmp = (struct Node\*) mallocMemory(sizeof(struct Node));

tmp-&gt;from = from;

tmp-&gt;size = size;

tmp-&gt;next = node-&gt;next;

node-&gt;next = tmp;

return;

}

insertSorted(node-&gt;next, from, size);

}

**// Insert At the Head of the Linked List**

struct Node\* insertAtHead(struct Node \*node, int from, int size){

struct Node \*tmp = (struct Node\*) mallocMemory(sizeof(struct Node));

tmp-&gt;from = from;

tmp-&gt;size = size;

tmp-&gt;next = node;

return tmp;

}

**// Deleting at the Given Index**

struct Node \*deleteAtIndex(struct Node \*node, int index){

int c = 1;

while (1){

if (c == index){

struct Node \*tmp = node-&gt;next;

node-&gt;next = node-&gt;next- &gt;next;

return tmp;

}

c++;

node = node-&gt;next;

}

}

// **Printing the Memory Allocations**

void printMemory(struct Node \*node, int from){

if (!node){

return;

}

if (from){

printMemoryf(&quot;%d &quot;, node-&gt;from);

}

printMemoryf(&quot;%d\n&quot;, node-&gt;size);

printMemory(node-&gt;next, from);

}

// **Combining the memory Allocations**

void combine(struct Node \*node){

if (node-&gt;size == 0){

if (node-&gt;next){

node-&gt;from = node-&gt;next- &gt;from;

node-&gt;size = node-&gt;next- &gt;size;

node-&gt;next = node-&gt;next- &gt;next;

}

else{

freeMemory = NULL;

}

}

else if (node-&gt;next &amp;&amp; node-&gt;from + node-&gt;size == node-&gt;next- &gt;from){

node-&gt;size += node-&gt;next- &gt;size;

node-&gt;next = node-&gt;next- &gt;next;

combine(node);

}

else if(node-&gt;next){

combine(node-&gt;next);

}

}

int length(struct Node \*node){

int l = 0;

while (node){

l++;

node = node-&gt;next;

}

return (l);

}

//**1. First Fit: In the first fit, partition is allocated which is first**

**// sufficient from the top of Main Memory.**

struct Node \*first\_fit(struct Node \*node, int size){

while (node){

if (node-&gt;size &gt;= size &amp;&amp; size &gt; 0){

return node;

}

else{

node = node-&gt;next;

}

}

return (NULL);

}

//**2. Best Fit : Allocate the process to the partition which is first**

**// smallest sufficient partition among the free available partition.**

struct Node \*best\_fit(struct Node \*node, int size){

int min = 1000;

struct Node \*tmp = NULL;

while (node){

if (node-&gt;size - size &lt; min &amp;&amp; node-&gt;size - size &gt;= 0 &amp;&amp; size &gt; 0){

min = node-&gt;size - size;

tmp = node;

}

else{

node = node-&gt;next;

}

}

return (tmp);

}

//**3.Worst Fit : Allocate the process to the partition which is largest**

**// sufficient among the freely available partitions available in**

**// the main memory.**

struct Node \*worst\_fit(struct Node \*node, int size){

int max = -1;

struct Node \*tmp = NULL;

while (node){

if (node-&gt;size - size &gt; max &amp;&amp; node-&gt;size - size &gt;= 0 &amp;&amp; size &gt; 0){

max = node-&gt;size - size;

tmp = node;

}

else{

node = node-&gt;next;

}

}

return (NULL);

}

main(){

printMemoryf(&quot;\nTotal memory size: 100kb\n\n&quot;); //Intialising the

Memory

freeMemory = insertAtHead(NULL, 1, 100);

int loop = 1;

while (loop) {

combine(freeMemory); // combine free spaces

printMemory(freeMemory, 1); // printMemory free memory

int c;

printMemoryf(&quot;\n\n1. New process\n2. Stop an existing

process\n3. Exit\nYour choice: &quot;);

scanf(&quot;%d&quot;, &amp;c); // Scanning the User Input

switch(c) {

case 1:

printMemoryf(&quot;\nEnter the size of process: &quot;);

int s;

scanf(&quot;%d&quot;, &amp;s);

printMemoryf(&quot;\nWhich fit to use?\n1. First fit\n2. Best fit\n3.

Worst fit\n$ &quot;);

int f;

scanf(&quot;%d&quot;, &amp;f);

struct Node \*r = NULL;

switch(f){

case 1:

r = first\_fit(freeMemory, s); // Calling First Fit

Memory Allocation

break;

case 2:

r = best\_fit(freeMemory, s); // Calling Best Fit

Memory Allocation

break;

case 3:

r = worst\_fit(freeMemory, s); // Calling Worst Fit Memory

Allocation

break;

default:

printMemoryf(&quot;\nInvalid choice\n&quot;);

break;

}

if (!r) {

printMemoryf(&quot;\nMemory could not be

allocated\n\n&quot;);

}

else { // Allocating the

memory

allocMemory = insertAtHead(allocMemory, r-&gt;from, s);

r-&gt;from += s;

r-&gt;size -= s;

printMemoryf(&quot;Memory allocMemoryated\n\n&quot;);}

break;

case 2:

if (length(allocMemory) &gt; 0){

printMemoryf(&quot;\nRunning processes:\n&quot;);

printMemory(allocMemory, 0);

printMemoryf(&quot;\n\nEnter index of process to be stopped: &quot;);

int p;

scanf(&quot;%d&quot;, &amp;p);

if (p==0){ // Stop process

insertSorted(freeMemory, allocMemory-&gt;from, allocMemory-&gt;size);

allocMemory = allocMemory-&gt;next;

}

else{

struct Node \*tmp = deleteAtIndex(allocMemory, p);

if (freeMemory-&gt;from &lt; tmp-&gt;from){

insertSorted(freeMemory, tmp-&gt;from, tmp-&gt;size);

}

else{

freeMemory = insertAtHead(freeMemory, tmp-&gt;from, tmp-&gt;size);

}

}

}

else{

printMemoryf(&quot;\nNo running processes!\n&quot;);

}

break;

case 3:

loop = 0;

break;

default:

printMemoryf(&quot;\nInvalid input\n\n&quot;);

break;

}

}

}

**LAB 2**

**AIM:**

To simulate paging in dynamic memory allocation using linked lists to maintain reference to the constituent pages of a process.

**THEORY:**

The downside to contiguous memory allocation is the external fragmentation that takes place, ie free spans of memory lie within allocated blocks.

Paging is an alternative to the contiguous allocation schemes, that removes the problem of external fragmentation. It involves division if the process into fixed sized “Pages”, or “Frames” within the secondary memory. Partitions of the main memory are page sized, and thus pages are loaded, not always the entire process at a time. Each page is allocated separately, and thus memory for a given process is allocated non contiguously.

Linked lists between the pages serve as a way of keeping track of the other pages of a process with minimal extra space required. However, it is slower, and Paging as a whole is subject to internal fragmentation within the pages, when the size allocated to a page is less than the page size.

**METHOD:**

The Paging system is simulated in JAVA, with “page” objects being assigned locations within the main memory (simulated using an array of pages). Each page object has an attribute “next\_page”, which contains a reference to the next page of the process, and thus given a page within the memory, we can reach all others of the same process.

Page size is hard coded into the system, but number of processes and their sizes are read through user input.

**CODE:**

**1. Using Linked Lists**.

import java.util.\*;

class Frame{ // **Used to define Frames**

int allocated;

Page page;

Frame(int i){

this.allocated=i;

}

}

class Page{ // **Used to define Pages**

int process\_id;

int page\_no;

int address=-1;

Page(int id,int page\_no){

this.process\_id=id;

this.page\_no=page\_no;

}

}

class Process{ // **Used to define Process**

int id;

int size;

LinkedList&lt;Page&gt; pages=new LinkedList&lt;Page&gt;(); //**Define linked list of pages**

LinkedList&lt;Frame&gt; frames=new LinkedList&lt;Frame&gt;(); //**Define linked list of frames**

private Page page;

Process(int id,int a){

this.id=id;

this.size=a;

int p=(this.size/page\_list.p) + (((this.size%page\_list.p)&gt;0)?1:0);

for(int i=0;i&lt;p;i++){

page=new Page(id,i+1);

pages.add(page);

}

}

void allocate(){ //**Allocates pages to frames**

for(Page p: pages){

if(p.address==-1)

for(int i=0;i&lt;page\_list.frame.length;i++)

if(page\_list.frame[i].allocated==0){

page\_list.frame[i].allocated=1;

page\_list.frame[i].page=p;

p.address=i;

this.frames.add(page\_list.frame[i]);

break;

}

}

}

}

class page\_list{ //**Main class**

static int p;

static Frame []frame; //**Array of Frames**

static List&lt;Process&gt; process\_list; //**Linked list of processes**

public static void main(String args[]){

int process\_count=0;

process\_list=new LinkedList&lt;Process&gt;();

Scanner sc=new Scanner(System.in);

System.out.println(&quot;Enter size of pages.&quot;);

p=sc.nextInt();

System.out.println(&quot;Enter total no. of frames in memory.&quot;);

int total\_frames=sc.nextInt();

frame=new Frame [total\_frames];

for(int i=0;i&lt;total\_frames;i++){

frame[i]=new Frame (0);

}

int choice=1;

Process pr;

while(choice!=0){ //**Infinite while loop for choices**

System.out.println(&quot;The Frames are:\nFrame No.\tProcess ID\t Page

No.&quot;);

int it=0;

for(Frame f:frame){

System.out.print(it+&quot;\t\t&quot;);

it++;

if(f.allocated==1)

System.out.println(f.page.process\_id+&quot;\t\t

&quot;+f.page.page\_no);

else

System.out.println(&quot;free \t\t free&quot;);

}

System.out.println(&quot;\n1.Add Process.\n2.Remove process.\n0.Exit&quot;);

choice=sc.nextInt();

switch(choice){

case 1: //**Adding a new process**

process\_count++;

System.out.println(&quot;Enter Size of process.&quot;);

pr=new Process(process\_count,sc.nextInt());

process\_list.add(pr);

pr.allocate();

break;

case 2: //**Removing a process**

System.out.println(&quot;Processes:&quot;);

for(int i=0;i&lt;process\_list.size();i++){

System.out.println(i+&quot; : &quot;+process\_list.get(i).size);

}

int delete\_index=sc.nextInt();

Process delete\_process=process\_list.get(delete\_index);

for(Frame f: delete\_process.frames){

f.allocated=0;

f.page=null;

}

process\_list.remove(delete\_index);

for(int i=delete\_index;i&lt;process\_list.size();i++){

process\_list.get(i).allocate();

}

default: break;

}

}

}

}

**2. Using Page Table.**

import java.util.\*;

class Frame{ **// Used to define Frames**

int allocated;

Page page;

Frame(int i){

this.allocated=i;

}

}

class Page{ // **Used to define Pages**

int process\_id;

int page\_no;

int address=-1;

Page(int id,int page\_no){

this.process\_id=id;

this.page\_no=page\_no;

}

}

class Page\_Table{ //**Used to define page table of process**

int process\_id;

Page[] pages;

Page\_Table(int i){

pages=new Page[i];

}

}

class Process{ // **Used to define Process**

int id;

int size;

Page\_Table PT; //**Defining Page Table**

LinkedList&lt;Frame&gt; frames=new LinkedList&lt;Frame&gt;(); //**Define linked list of frames**

private Page page;

Process(int id,int a){

this.id=id;

this.size=a;

int p=(this.size/page\_table.p) + (((this.size%page\_table.p)&gt;0)?1:0);

PT=new Page\_Table(p);

for(int i=0;i&lt;p;i++){

PT.pages[i]=new Page(id,i+1);

}

}

void allocate(){ //**Allocates pages to frames**

for(Page p: PT.pages){

if(p.address==-1)

for(int i=0;i&lt;page\_table.frame.length;i++)

if(page\_table.frame[i].allocated==0){

page\_table.frame[i].allocated=1;

page\_table.frame[i].page=p;

p.address=i;

this.frames.add(page\_table.frame[i]);

break;

}

}

}

}

class page\_table{ //**Main class**

static int p;

static Frame []frame; //**Array of Frames**

static List&lt;Process&gt; process\_list; //**Linked list of processes**

public static void main(String args[]){

int process\_count=0;

process\_list=new LinkedList&lt;Process&gt;();

Scanner sc=new Scanner(System.in);

System.out.println(&quot;Enter size of pages.&quot;);

p=sc.nextInt();

System.out.println(&quot;Enter total no. of frames in memory.&quot;);

int total\_frames=sc.nextInt();

frame=new Frame [total\_frames];

for(int i=0;i&lt;total\_frames;i++){

frame[i]=new Frame (0);

}

int choice=1;

Process pr;

while(choice!=0){ //**Infinite while loop for**

choices

System.out.println(&quot;The Frames are:\nFrame No.\tProcess ID\t Page

No.&quot;);

int it=0;

for(Frame f:frame){

System.out.print(it+&quot;\t\t&quot;);

it++;

if(f.allocated==1)

System.out.println(f.page.process\_id+&quot;\t\t

&quot;+f.page.page\_no);

else

System.out.println(&quot;free \t\t free&quot;);

}

System.out.println(&quot;\n1.Add Process.\n2.Remove process.\n0.Exit&quot;);

choice=sc.nextInt();

switch(choice){

case 1: //**Adding a new**

process

process\_count++;

System.out.println(&quot;Enter Size of process.&quot;);

pr=new Process(process\_count,sc.nextInt());

process\_list.add(pr);

pr.allocate();

break;

case 2: //**Removing a process**

System.out.println(&quot;Processes:&quot;);

for(int i=0;i&lt;process\_list.size();i++){

System.out.println(i+&quot; : &quot;+process\_list.get(i).size);

}

int delete\_index=sc.nextInt();

Process delete\_process=process\_list.get(delete\_index);

for(Frame f: delete\_process.frames){

f.allocated=0;

f.page=null;

}

process\_list.remove(delete\_index);

for(int i=delete\_index;i&lt;process\_list.size();i++){

process\_list.get(i).allocate();

}

default: break;

}

}

}

}

**LAB 3**

**AIM:**

To simulate the removal of external fragmentation in case of contiguous memory allocation with variable size partitioning.

**THEORY:**

One of the disadvantages of using a contiguous memory allocation scheme is the occurrence of external fragmentation of the memory, where small blocks of free memory remain scattered between allocated blocks. These blocks may not be appropriately sized to house processes on their own, but if combined can often allow for more processes to be allocated.

Thus, removal of external fragmentation is a valuable thing to consider during design of memory allocation schemes, and is usually accomplished by accumulation of all free blocks to one end of the memory range of main memory (either the start or end).

However, in practice it can prove to be very costly, as often large blocks will have to be moved to allow for small unallocated space to move to the end.

**METHOD:**

In this C simulation, main memory is maintained in a linked list with a sentinel node, which houses the free memory. Allocated memory blocks are assigned using malloc and linked to the sentinel node. Deallocation allows for the freed space to be added to the sentinel node block directly, accumulating the free space towards the front end of the memory.

**CODE:**

#include&lt;stdio.h&gt;

#include&lt;stdlib.h&gt;

// **Linked list to keep track of allocated blocks**

struct Node

{

int start, end, number, size;

struct Node \*next;

};

// **Insert a node when a new process is allocated**

void insert\_allocated(struct Node \*\*head, int size, int start)

{

// ‘**size’ is the size of process, ‘start’ is the starting address of ‘free’ linked list before**

**// allocating the new process**

struct Node\* new\_node = NULL;

new\_node = (struct Node\*)malloc(sizeof(struct Node));

new\_node -&gt; start = start;

new\_node -&gt; end = start + size;

new\_node -&gt; size = size;

if((\*head) == NULL) // **check if head is null**

{

(\*head) = new\_node;

return;

}

struct Node\* pointer1 = NULL;

pointer1 = (struct Node\*)malloc(sizeof(struct Node));

pointer1 = (\*head);

struct Node\* pointer2 = NULL;

pointer2 = (struct Node\*)malloc(sizeof(struct Node));

pointer2 = (\*head);

pointer1 = pointer1 -&gt; next;

while(pointer1 != NULL) // **insert node at the tail of linked list**

{

pointer2 = pointer1;

pointer1 = pointer1 -&gt; next;

}

pointer2 -&gt; next = new\_node;

}

int move\_allocated(struct Node \*\*head, int num\_process)

{

int free\_start;

int current\_proc = 1;

struct Node\* pointer = NULL;

pointer = (struct Node\*)malloc(sizeof(struct Node));

pointer = (\*head);

struct Node\* pointer\_prev = NULL;

pointer\_prev = (struct Node\*)malloc(sizeof(struct Node));

pointer\_prev = (\*head);

while(current\_proc != num\_process) **// search for process ‘num\_process’**

{

if(current\_proc = num\_process - 1)

pointer\_prev = pointer;

pointer = pointer -&gt; next;

current\_proc++;

}

pointer\_prev -&gt; next = pointer -&gt; next;

int size\_current = pointer -&gt; size;

pointer = pointer -&gt; next;

while(pointer != NULL)

{

if(pointer -&gt; next == NULL)

{

free\_start = pointer -&gt; end; **// change start address of ‘free’ linked**

list

}

pointer -&gt; start -= size\_current; **// shift start and end address of ‘allocated’**

pointer -&gt; end -= size\_current;

pointer = pointer -&gt; next;

}

return size\_current;

}

// **print linked list**

void print(struct Node \*\*head)

{

struct Node\* pointer = NULL;

pointer = (struct Node\*)malloc(sizeof(struct Node));

pointer = (\*head);

while(pointer != NULL)

{

printf(&quot;%d %d\n&quot;, pointer -&gt; start, pointer -&gt; end);

pointer = pointer -&gt; next;

}

}

int main()

{

int n;

printf(&quot;Enter number of processes:&quot;);

scanf(&quot;%d&quot;, &amp;n);

int a[1024]; // **array which stores processes.**

struct Node\* free = NULL;

struct Node\* allocated = NULL;

free = (struct Node\*)malloc(sizeof(struct Node)); // **Linked list for free memory**

allocated = (struct Node\*)malloc(sizeof(struct Node)); **// Linked list for allocated**

memory

free -&gt; start = 0; // Initially, whole memory is free

free -&gt; end = 1024;

free -&gt; next = NULL;

printf(&quot;Enter size of %d processes:\n&quot;, n);

for(int i = 0; i &lt; n; i++)

{

int size;

scanf(&quot;%d&quot;, &amp;size);

int ref = free -&gt; start;

free -&gt; start += size; // **increment start address of ‘free’ linked list by size**

of // process

insert\_allocated(&amp;allocated, size, ref); // **insert a node for allocated**

memory

}

printf(&quot;Allocated Linked list:\n&quot;);

print(&amp;allocated);

printf(&quot;Free linked list:\n&quot;);

print(&amp;free);

printf(&quot;Enter number of processes to delete:&quot;);

int num\_del;

scanf(&quot;%d&quot;, &amp;num\_del);

printf(&quot;Enter process numbers:\n&quot;);

for(int i = 0; i &lt; num\_del; i++)

{

int num\_process;

scanf(&quot;%d&quot;, &amp;num\_process);

int free\_start = move\_allocated(&amp;allocated, num\_process); **// make memory**

**// allocation contiguous by calling the function ‘move\_allocated’**

printf(&quot;%d\n&quot;, free\_start);

free -&gt; start -= free\_start; // **change start address of ‘free’ linked list**

}

printf(&quot;Allocated Linked list:\n&quot;);

print(&amp;allocated);

printf(&quot;Free linked list:\n&quot;);

print(&amp;free);

printf(&quot;Enter number of processes to add:&quot;);

int add\_process;

scanf(&quot;%d&quot;, &amp;add\_process);

printf(&quot;Enter size of processes to add:\n&quot;);

for(int i = 0; i &lt; add\_process; i++)

{

int size;

scanf(&quot;%d&quot;, &amp;size);

int ref = free -&gt; start;

free -&gt; start += size;

insert\_allocated(&amp;allocated, size, ref); // **insert a node for allocated**

memory

}

printf(&quot;Allocated Linked list:\n&quot;);

print(&amp;allocated);

printf(&quot;Free linked list:\n&quot;);

print(&amp;free);

}

**LAB 4**

**AIM:**

To simulate buddy memory allocation scheme, and understand allocation, deletion and merging of memory blocks.

**THEORY:**

The buddy memory allocation technique is a memory allocation algorithm that divides memory into partitions to try to satisfy a memory request as suitably as possible. This system makes use of splitting memory into halves to try to give a best fit. Every memory block in this system has an order, where the order is an integer ranging from 0 to a specified upper limit.

The size of a block of order n is proportional to 2n, so that the blocks are exactly twice the size of blocks that are one order lower. Power-of-two block sizes make address computation simple, because all buddies are aligned on memory address boundaries that are powers of two. When a larger block is split, it is divided into two smaller blocks, and each smaller block becomes a unique buddy to the other. A split block can only be merged with its unique buddy block, which then reforms the larger block they were split from.

**METHOD:**

In this simulation in C, the blocks are maintained in an array implementation of a binary tree. Each block contains attributes to determine if it is free, or has been split into 2 child blocks. Allocation scheme favours allocation of the left child over the right child, and a recursive allocation as well as merge function allow for tree traversal.

Merging occurs only when both sibling blocks are free, and is a recursive process up the tree until the root, at index zero is reached.

For allocation, the minimum size required is determined and the tree is traversed to determine if the block of that size is free. Splitting of nodes occurs along the way.

**CODE:**

#include&lt;stdio.h&gt;

#include&lt;conio.h&gt;

int tree[2050], i, j, k;

int place (int), power (int, int);

void segmentalloc (int totsize, int request) {

int flevel = 0, size;

size = totsize;

if (request &gt; totsize) { //**size of the process is more than total**

memory

printf (&quot; RESULT : \n&quot;);

printf (&quot;The system don&#39;t have enough free memory\n&quot;);

printf (&quot;Suggession : Go for VIRTUAL MEMORY\n&quot;);

getch ();

return;

}

while (1) {

if (request &lt;= size &amp;&amp; request &gt; (size / 2))

break;

else{

size /= 2;

flevel++;

}

}

for (i = power (2, flevel) - 1; i &lt;= (power (2, flevel + 1) - 2); i++)

if (tree[i] == 0 &amp;&amp; place (i)) {

tree[i] = request;

makedivided (i);

printf (&quot;Result: Successful Allocation\n&quot;);

break;

}

if (i == power (2, flevel + 1) - 1) { //**size of the process is more than available** **size**

printf (&quot;Result : \n&quot;);

printf (&quot;The system don&#39;t have enough free memory\n&quot;);

printf (&quot;Suggession : Go for VIRTUAL Memory Mode\n&quot;);

}

}

void makedivided (int node) { //**Dividing the memory into halves untill the process is best fit**

while (node != 0)

{

node = node % 2 == 0 ? (node - 1) / 2 : node / 2;

tree[node] = 1;

}

}

int place (int node) { //**Checking for available nodes in a particular level**

while (node != 0) {

node = node % 2 == 0 ? (node - 1) / 2 : node / 2;

if (tree[node] &gt; 1)

return 0;

}

return 1;

}

void makefree (int request){ //**Deleting a process from the memory**

int node = 0;

while (1) {

if (tree[node] == request)

break;

else

node++;

}

tree[node] = 0;

while (node != 0) {

if (tree[node % 2 == 0 ? node - 1 : node + 1] == 0 &amp;&amp; tree[node] == 0) {

tree[node % 2 == 0 ? (node - 1) / 2 : node / 2] = 0;

node = node % 2 == 0 ? (node - 1) / 2 : node / 2;

}

else

break;

}

}

int power (int x, int y) { //**computing x power y**

int z, ans;

if (y == 0)

return 1;

ans = x;

for (z = 1; z &lt; y; z++)

ans \*= x;

return ans;

}

void printing (int totsize, int node) { //**printing memory map**

int permission = 0, llimit, ulimit, tab;

if (node == 0)

permission = 1;

else if (node % 2 == 0)

permission = tree[(node - 1) / 2] == 1 ? 1 : 0;

else

permission = tree[node / 2] == 1 ? 1 : 0;

if (permission) {

llimit = ulimit = tab = 0;

while (1) {

if (node &gt;= llimit &amp;&amp; node &lt;= ulimit)

break;

else {

tab++;

printf (&quot; &quot;);

llimit = ulimit + 1;

ulimit = 2 \* ulimit + 2;

}

}

//**recursively printing for each level if it is divided or allocated or free**

printf (&quot; %d &quot;, totsize / power (2, tab));

if (tree[node] &gt; 1)

printf (&quot;-- -&gt; Allocated %d\n&quot;, tree[node]);

else if (tree[node] == 1)

printf (&quot;-- -&gt; Divided\n&quot;);

else

printf (&quot;-- -&gt; Free\n&quot;);

//printing memory map for next sub levels

printing (totsize, 2 \* node + 1);

printing (totsize, 2 \* node + 2);

}

}

int main () //**The main function**

{

int totsize, cho, req;

printf (&quot;Enter the Size of the Memory : \n&quot;); //**Total Size of Memory(Generally in**

**// powers of 2)**

scanf (&quot;%d&quot;, &amp;totsize);

while (1) { //Infinite while loop for choices

printf (&quot;BUDDY SYSTEM\n&quot;);//**Menu**

printf (&quot;1) Add a process into the Memory\n&quot;);//**Add**

printf (&quot;2) Remove the process from Memory\n&quot;);//**Delete**

printf (&quot;3) Tree structure for Memory allocation Map\n&quot;);//**Memory Map**

printf (&quot;4) Exit\n&quot;);

printf (&quot; \* Enter your choice : \n&quot;);

scanf (&quot;%d&quot;, &amp;cho);

switch (cho){

case 1: //**Allocating a process**

printf (&quot; MEMORY ALLOCATION \n&quot;);

printf (&quot;Enter the Process size : \n&quot;);

scanf (&quot;%d&quot;, &amp;req);

segmentalloc (totsize, req);//**Function Call for adding a process**

break;

case 2: //**Deallocating a process**

printf(&quot;MEMORY DEALLOCATION \n&quot;);

printf (&quot;\* Enter the process size : \n&quot;);

scanf (&quot;%d&quot;, &amp;req);

makefree (req);//**Function Call for removing a process**

break;

case 3: //**Printing the buddy tree**

printf (&quot;MEMORY ALLOCATION MAP\n&quot;);

printing (totsize, 0);//**Function Call for printing mamory map**

getch ();

break;

default:

return;

}

}

return 0;

}