DAYANANDA SAGAR UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
SCHOOL OF ENGINEERING

DAYANANDA SAGAR UNIVERSITY
KUDLU GATE

BANGALORE - 560068

MINI PROJECT REPORT

ON

“BUDDY SYSTEM”

SUBMITTED TO THE Vth SEMESTER OPERATING SYSTEM LABORATORY-2019

BACHELOR OF TECHNOLOGY

IN

COMPUTER SCIENCE & ENGINEERING

Submitted by

KIRAM M-(ENG17CS0111)
K.DEVA BHAGRAV-(ENG17CS0107)
GOWTHAM R-(ENG17CS0078)
GOWTHAM B-(ENG17CS0077)

Under the supervision of

PRAMOD T.C

DAYANANDA SAGAR UNIVERSITY

School of Engineering, Kudlu Gate, Bangalore-560068

CERTIFICATE

This is to certify that Mr./Ms. ______________________________ bearing USN ________________ has satisfactorily completed his/her Mini Project as prescribed by the University for the __________ semester B.Tech. programme in Computer Science & Engineering during the year __________ at the School of Engineering, Dayananda Sagar University., Bangalore.

Date: ____________

____
Signature of the faculty in-charge

Max Marks	Marks Obtained

____

Signature of Chairman
Department of Computer Science & Engineering

DECLARATION

We hereby declare that the work presented in this mini project entiltled-"BUDDY SYSTEM ", has been carried out by us and it has not been submitted for the award of any degree,diploma or the mini project of any other college or university.

KIRAM M-(ENG17CS0111)
K.DEVA BHAGRAV-(ENG17CS0107)
GOWTHAM R-(ENG17CS0078)
GOWTHAM B-(ENG17CS0077)

ACKNOWLEDGEMENT

This satisfaction that accompanies the successful compilation of any task would be incomplete without the mention of people who made it possible and whose constant guidance and encouragement crown all the efforts with success.

We are especially thankful to our CHAIRMAN Dr. BANGA M K, Ph.D., for providing necessary departmental facilities and moral support and encouragement.

We are very much thankful to our Associate Professor Department of CSE **Mr**,TC PRAMOD for initiating us into this project by providing information regarding this project, guidance, support, motivation and patience which helped us for successful completion of our mini project.

We have received a great deal of guidance and co-operation from our friends and we wish to thank one and all that directly or indirectly helped us in the successful completion of this project work.

KIRAM M-(ENG17CS0111)
K.DEVA BHAGRAV-(ENG17CS0107)
GOWTHAM R-(ENG17CS0078)
GOWTHAM B-(ENG17CS0077)

CONTENTS

ABSTRACT

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. The Buddy memory allocation is relatively easy to implement. It supports limited but efficient splitting and coalescing of memory blocks.

Buddy system allows a single allocation block to be split, to form two blocks half the size of the parent block. These two blocks are known as 'buddies'. Part of the definition of a 'buddy' is that the buddy of block *B* must be the same size as *B*, and must be adjacent in memory (so that it is possible to merge them later).

The other important property of buddies, stems from the fact that in the buddy system, every block is at an address in memory which is exactly divisible by its size. So all the 16-byte blocks are at addresses which are multiples of 16; all the 64K blocks are at addresses which are multiples of 64K... and so on.

Not only must buddies be adjacent in memory, but the lower 'buddy' must be at a location divisible by their combined size. For example, of two 64K blocks, they are only buddies if the lower block lies at an address divisible by 128K. This ensures that if they are merged, the combined block maintains the property described above.

INTRODUCTION

ABOUT THE PROBLEM

Buddy system of memory management attempts to be fast at allocating block of correct size and also, easy to merge adjacent holes. (We saw when you sort a free list by block size that allocations are fast, but merging is very difficult.) Exploits fact that computers deal easily with powers of two.

We create several free block lists, each for a power-of-two size.

So, for example, if the minimum allocation size is 8 bytes, and the memory size is 1MB, we create a list for 8 bytes hole, a list for 16 byte holes, one for 32-bytes holes, 64, 128, 256, 512, 1K, 2K, 4K, 8K, 16K, 32K, 64K, 128K, 256K, 512K and one list for 1MB holes.

All the lists are initially empty, except for the 1MB list, which has one hole listed.

All allocations are rounded up to a power of two---70K allocations rounded up to 128K, 15K allocations rounded up to 16K, etc.

OS TECHNIQUE

MEMORY MANAGEMENT

- **Allocation technique:** Memory allocation is the process of assigning blocks of memory on request. Typically the allocator receives memory from the operating system in a small number of large blocks that it must divide up to satisfy the requests for smaller blocks. It must also make any returned blocks available for reuse. There are many common ways to perform this, with different strengths and weaknesses. A few are described briefly below.

PROBLEM STATEMENT

Buddy system of memory management attempts to be fast at allocating block of correct size and also, easy to merge adjacent holes. (We saw when you sort a free list by block size that allocations are fast, but merging is very difficult.) Exploits fact that computers deal easily with powers of two.

We create several free block lists, each for a power-of-two size.

So, for example, if the minimum allocation size is 8 bytes, and the memory size is 1MB, we create a list for 8 bytes hole, a list for 16 byte holes, one for 32-bytes holes, 64, 128, 256, 512, 1K, 2K, 4K, 8K, 16K, 32K, 64K, 128K, 256K, 512K and one list for 1MB holes.

All the lists are initially empty, except for the 1MB list, which has one hole listed.

All allocations are rounded up to a power of two---70K allocations rounded up to 128K, 15K allocations rounded up to 16K, etc.

LITERATURE REVIEW

There are various forms of the buddy system; those in which each block is subdivided into two smaller blocks are the simplest and most common variety. 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 2^n, 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.

Starting off, the size of the smallest possible block is determined, i.e. the smallest memory block that can be allocated. If no lower limit existed at all (e.g., bit-sized allocations were possible), there would be a lot of memory and computational overhead for the system to keep track of which parts of the memory are allocated and unallocated. However, a rather low limit may be desirable, so that the average memory waste per allocation (concerning allocations that are, in size, not multiples of the smallest block) is minimized. Typically the lower limit would be small enough to minimize the average wasted space per allocation, but large enough to avoid excessive overhead. The smallest block size is then taken as the size of an order-0 block, so that all higher orders are expressed as power-of-two multiples of this size.

Starting off, the size of the smallest possible block is determined, i.e. the smallest memory block that can be allocated. If no lower limit existed at all (e.g., bit-sized allocations were possible), there would be a lot of memory and computational overhead for the system to keep track of which parts of the memory are allocated and unallocated. However, a rather low limit may be desirable, so that the average memory waste per allocation (concerning allocations that are, in size, not multiples of the smallest block) is minimized. Typically the lower limit would be small enough to minimize the average wasted space per allocation, but large enough to avoid excessive overhead. The smallest block size is then taken as the size of an order-0 block, so that all higher orders are expressed as power-of-two multiples of this size.

The programmer then has to decide on, or to write code to obtain, the highest possible order that can fit in the remaining available memory space. Since the total available memory in a given computer system may not be a power-of-two multiple of the minimum block size, the largest block size may not span the entire memory of the system. For instance, if the system had 2000 K of physical memory and the order-0 block size was 4 K, the upper limit on the order would be 8, since an order-8 block (256 order-0 blocks, 1024 K) is the biggest block that will fit in memory. Consequently it is impossible to allocate the entire physical memory in a single chunk; the remaining 976 K of memory would have to be allocated in smaller blocks.

SOFTWARE AND HARDWARE REQUIREMENTS

- HARDWARE
  - LAPTOP(PC)
- SOFTWARE
  - C COMPILER

DESIGN

PSEUDOCODE OF ALGORITHM

```
void segmentalloc(int totsize,int request)
{
	int flevel=0,size;
	size=totsize;
	if(request>totsize)
	{
		printf("%d  R E S U L T  : \t ",2);
		printf("*  The system don't have enough free memory\n");
		printf("*  Suggession  :  Go for VIRTUAL MEMORY\n");
		return;
	}
	while(1)
	{
		if(request<size && request>(size/2))
			break;
		else
		{
			size/=2;
			flevel++;
		}
	}
	for(i=power(2,flevel)-1;i<=(power(2,flevel+1)-2);i++)
		if(tree[i]==0 && place(i))
		{
			tree[i]=request;
			makedivided(i);
			printf(" Result    :     Successful Allocation\n");
			break;
		}
	if(i==power(2,flevel+1)-1)
	{
		printf("    Result  :  \t");
		printf("*  The system don't have enough free memory\n");
		printf("*  Suggession  :  Go for VIRTUAL Memory Mode\n");
	}
}

void makedivided(int node)
{
	while(node!=0)
	{
		node=node%2==0?(node-1)/2:node/2;
		tree[node]=1;
```

```
	}
}

int place(int node)
{
	while(node!=0)
	{
		node=node%2==0?(node-1)/2:node/2;
		if(tree[node]>1)
			return 0;
	}
	return 1;
}

void makefree(int request)
{
	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 && 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)
{
	int z,ans;
	if(y==0) return 1;
	ans=x;
	for(z=1;z<y;z++)
		ans*=x;
	return ans;
}

void printing(int totsize,int node)
{
	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>=llimit && node<=ulimit)
				break;
			else
			{
				tab++;
				printf("   ");
				llimit=ulimit+1;
				ulimit=2*ulimit+2;
			}
		}
		printf(" %d ",totsize/power(2,tab));
		if(tree[node]>1)
			printf("---> Allocated %d \n",tree[node]);
		else if(tree[node]==1)
			printf("---> Divided ");
		else printf("---> Free ");
		printing(totsize,2*node+1);
		printing(totsize,2*node+2);
	}
}
```

IMPLEMENTATION

```
//                    BUDDY SYSTEM CODE

#include<stdio.h>

int tree[2050],i,j,k;
void segmentalloc(int,int),makedivided(int),makefree(int),printing(int,int);
int place(int),power(int,int);
int main()
{
        int totsize,cho,req;

        printf(" B U D D Y   S Y S T E M  R E Q U I R E M E N T S\n");
        printf("        *  Enter the Size of the memory  : \t ");
        scanf("%d",&totsize);
        while(1)
        {
                printf("\n B U D D Y   S Y S T E M \n");
                printf("*  1)   Locate the process into the Memory\n");
                printf("*  2)   Remove the process from Memory\n");
                printf("*  3)   Tree structure for Memory allocation Map\n");
                printf("*  4)   Exit\n");
                printf("*  Enter your choice  :  ");
                scanf("%d",&cho);
                switch(cho)
                {
                        case 1:

                                printf(" ");
                                printf(" ");
                                printf(" M E M O R Y   A L L O C A T I O N \n");
                                printf("*  Enter the Process size  : \t");
                                scanf("%d",&req);
                                segmentalloc(totsize,req);
                                break;
                        case 2:

                                printf(" ");

                                printf(" ");
                                printf(" M E M O R Y   D E A L L O C A T I O N \n");

                                printf("*  Enter the process size  : \t ");
                                scanf("%d",&req);
                                makefree(req);
                                break;
                        case 3:
```

```
                printf(" ");

                printf("M E M O R Y   A L L O C A T I O N   M A P\n");
                printf(" ");
                printing(totsize,0);
                printf(" ");
                break;
            default:
                return 0;
        }
    }
}

void segmentalloc(int totsize,int request)
{
    int flevel=0,size;
    size=totsize;
    if(request>totsize)
    {
        printf("%d  R E S U L T  : \t ",2);
        printf("*  The system don't have enough free memory\n");
        printf("*  Suggession  :  Go for VIRTUAL MEMORY\n");
        return;
    }
    while(1)
    {
        if(request<size && request>(size/2))
            break;
        else
        {
            size/=2;
            flevel++;
        }
    }
    for(i=power(2,flevel)-1;i<=(power(2,flevel+1)-2);i++)
        if(tree[i]==0 && place(i))
        {
            tree[i]=request;
            makedivided(i);
            printf(" Result    :     Successful Allocation\n");
            break;
        }
    if(i==power(2,flevel+1)-1)
    {
        printf("    Result  :  \t");
        printf("*  The system don't have enough free memory\n");
        printf("*  Suggession  :  Go for VIRTUAL Memory Mode\n");
    }
}

void makedivided(int node)
```

```
{
	while(node!=0)
	{
		node=node%2==0?(node-1)/2:node/2;
		tree[node]=1;
	}
}

int place(int node)
{
	while(node!=0)
	{
		node=node%2==0?(node-1)/2:node/2;
		if(tree[node]>1)
			return 0;
	}
	return 1;
}

void makefree(int request)
{
	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 && 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)
{
	int z,ans;
	if(y==0) return 1;
	ans=x;
	for(z=1;z<y;z++)
		ans*=x;
	return ans;
}
```

```
void printing(int totsize,int node)
{
	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>=llimit && node<=ulimit)
				break;
			else
			{
				tab++;
				printf("   ");
				llimit=ulimit+1;
				ulimit=2*ulimit+2;
			}
		}
		printf(" %d ",totsize/power(2,tab));
		if(tree[node]>1)
			printf("---> Allocated %d \n",tree[node]);
		else if(tree[node]==1)
			printf("---> Divided ");
		else printf("---> Free ");
		printing(totsize,2*node+1);
		printing(totsize,2*node+2);
	}
}
```

TESTING

.B U D D Y   S Y S T E M R E Q U I R E M E N T S
* Enter the Size of the memory :   200

B U D D Y   S Y S T E M
* 1)   Locate the process into the Memory
* 2)   Remove the process from Memory
* 3)   Tree structure for Memory allocation Map
* 4)   Exit
* Enter your choice : 1
M E M O R Y   A L L O C A T I O N
* Enter the Process size :   20
Result   :   Successful Allocation

B U D D Y   S Y S T E M
* 1)   Locate the process into the Memory
* 2)   Remove the process from Memory
* 3)   Tree structure for Memory allocation Map
* 4)   Exit
* Enter your choice : 3
M E M O R Y   A L L O C A T I O N   M A P
200 ---> Divided   100 ---> Divided   50 ---> Divided   25 ---> Allocated 20
25 ---> Free   50 ---> Free   100 ---> Free
B U D D Y   S Y S T E M
* 1)   Locate the process into the Memory
* 2)   Remove the process from Memory
* 3)   Tree structure for Memory allocation Map
* 4)   Exit
* Enter your choice : 4

B U D D Y   S Y S T E M R E Q U I R E M E N T S

* Enter the Size of the memory :   10

B U D D Y   S Y S T E M

* 1)   Locate the process into the Memory

* 2)   Remove the process from Memory

* 3)   Tree structure for Memory allocation Map

* 4)   Exit

* Enter your choice : 3

M E M O R Y   A L L O C A T I O N   M A P

10 ---> Free

B U D D Y   S Y S T E M

* 1)   Locate the process into the Memory

* 2)   Remove the process from Memory

* 3)   Tree structure for Memory allocation Map

* 4)   Exit

* Enter your choice : 4

OUTPUT SCREENSHOTS

```
 B U D D Y   S Y S T E M  R E Q U I R E M E N T S
        *  Enter the Size of the memory  :       1000

 B U D D Y   S Y S T E M
*  1)    Locate the process into the Memory
*  2)    Remove the process from Memory
*  3)    Tree structure for Memory allocation Map
*  4)    Exit
*  Enter your choice  :  1
   M E M O R Y   A L L O C A T I O N
*  Enter the Process size  :    200
 Result    :     Successful Allocation

 B U D D Y   S Y S T E M
*  1)    Locate the process into the Memory
*  2)    Remove the process from Memory
*  3)    Tree structure for Memory allocation Map
*  4)    Exit
*  Enter your choice  :  3
 M E M O R Y   A L L O C A T I O N   M A P
  1000 ---> Divided     500 ---> Divided        250 ---> Allocated 200
       250 ---> Free      500 ---> Free
 B U D D Y   S Y S T E M
*  1)    Locate the process into the Memory
*  2)    Remove the process from Memory
*  3)    Tree structure for Memory allocation Map
*  4)    Exit
*  Enter your choice  :  4


...Program finished with exit code 0
Press ENTER to exit console.
```

<u>PROCESS SIZE OF 200</u>

```
B U D D Y   S Y S T E M  R E Q U I R E M E N T S
       *  Enter the Size of the memory  :       100

B U D D Y   S Y S T E M
*  1)    Locate the process into the Memory
*  2)    Remove the process from Memory
*  3)    Tree structure for Memory allocation Map
*  4)    Exit
*  Enter your choice  :  1
   M E M O R Y   A L L O C A T I O N
*  Enter the Process size  :    200
2  R E S U L T  :        *  The system don't have enough free memory
*  Suggession  :  Go for VIRTUAL MEMORY

B U D D Y   S Y S T E M
*  1)    Locate the process into the Memory
*  2)    Remove the process from Memory
*  3)    Tree structure for Memory allocation Map
*  4)    Exit
*  Enter your choice  :
```

```
B U D D Y   S Y S T E M  R E Q U I R E M E N T S
       *  Enter the Size of the memory  :       100

B U D D Y   S Y S T E M
*  1)    Locate the process into the Memory
*  2)    Remove the process from Memory
*  3)    Tree structure for Memory allocation Map
*  4)    Exit
*  Enter your choice  :  1
   M E M O R Y   A L L O C A T I O N
*  Enter the Process size  :    20
Result    :     Successful Allocation

B U D D Y   S Y S T E M
*  1)    Locate the process into the Memory
*  2)    Remove the process from Memory
*  3)    Tree structure for Memory allocation Map
*  4)    Exit
*  Enter your choice  :  2
   M E M O R Y   D E A L L O C A T I O N
*  Enter the process size  :
```

DEALLOCATION

CONCLUSION

In this project we tried to simulate arrivals and departures from a fictional airport with a single runway which has a capacity of handling a total n aircraft at once (any mix of takeoffs and landings). The purpose of the simulation is to determine how long aircraft will wait to access the runway. Of particular interest is how long airplanes need to wait (queue) in the air before landing since that will require burning fuel, and this project shows the count of the number of airplanes landed and took off and crashed at some particular time and also it gives the total time took to perform landing, taking off and crashing operations took place and also total number of planes.

REFERENCES

https://www.pmi.org/learning/library/implementing-buddy-system-workplace-9376

https://www.geeksforgeeks.org/buddy-system-memory-allocation-technique/

https://www.memorymanagement.org/mmref/alloc.html