“Buddy System”

In Operating System

Ansari Arif(3117001)

Memon Shaziya(3117026)

Mishra Saloni(3117027)

Second Year Computer Engineering(Sem-IV)

Subject: Operating System(OS)

Prof. Farhana Siddiqui

Department of Computer Engineering

M. H. Saboo Siddik College of Engineering

8,Saboo Siddik Polytechnic Road, New Nagpada, Byculla,

Mumbai,Maharashtra 400008

2018

Index

Problem Statement:

Write a program to implement the buddy system of memory allocation in Operating Systems.

Theory:

Buddy allocation system is an algorithm in which a larger memory block is divided into small parts to satisfy the request. This algorithm is used to give best fit. The two smaller parts of block are of equal size and called as buddies. In the same manner one of the two buddies will further divide into smaller parts until the request is fulfilled. Benefit of this technique is that the two buddies can combine to form the block of larger size according to the memory request.

In buddy system, sizes of free blocks are in form of integral power of 2. E.g. 2, 4, 8, 16 etc. Up to the size of memory. When a free block of size 2k is requested, a free block from the list of free blocks of size 2k is allocated. If no free block of size 2k is available, the block of next larger size, 2k+1 is split in two halves called buddies to satisfy the request.

Why buddy system?

If the partition size and process size are different then poor match occurs and may use space inefficiently.
It is easy to implement and efficient then dynamic allocation.

Explanation

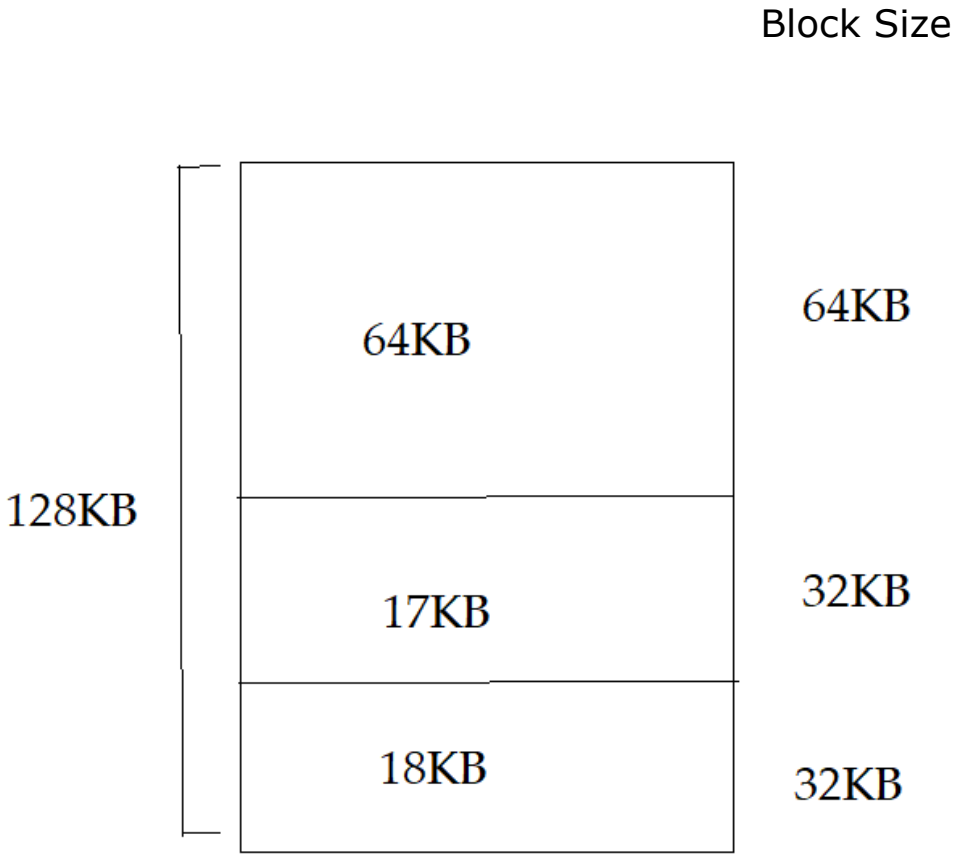
The buddy system is implemented as follows- A list of free nodes, of all the different possible powers of 2, is maintained at all times (So if total memory size is 1 MB, we'd have 20 free lists to track-one for blocks of size 1 byte, 1 for 2 bytes, next for 4 bytes and so on).

When a request for allocation comes, we look for the smallest block bigger than it. If such a block is found on the free list, the allocation is done (say, the request is of 27 KB and the free list tracking 32 KB blocks has at least one element in it), else we traverse the free list upwards till we find a big enough block. Then we keep splitting it in two blocks-one for adding to the next free list (of smaller size), one to traverse down the tree till we reach the target and return the requested memory block to the user. If no such allocation is possible, we simply return null.

Example:

Let total memory size be 128KB and let a process P1 is 18KB. 128KB block is split into two buddies of 64KB each, one is further split into two 32KB blocks and one of them is allocated to the process. Next P2 requires 40KB. 64KB block is allocated to 40KB.

Another example is let total memory size be 128KB and let a process P1 is 130KB, the system don't have enough free memory to occupy the process.



Allocating memory

To allocate, round the requested size up to a power of two (as mentioned), and search the appropriate list. If it is empty, allocate a block of twice the size, split it into two, add half to the free list, and use the other half. (This is recursive: the recursion stops when either we successfully find a block which we can split, or we find that we have reached the largest size of block and there are no free blocks available.)

So if our 1MB memory is initially empty, and we want to allocate a 70K block, we round it up to 128K, and end up splitting the 1MB into two ?meg blocks, splitting one of them into two 256K blocks, splitting one of them into two 128K blocks, and finally allocating one of the 128K blocks to the user:

1. Round 70K allocation to a power of 2: 128K.
2. Are there any 128K blocks available?
3. No. Allocate a 256K block.
    a. Are there any 256K blocks available?
    b. No. Allocate a 512K block.
        i. Are there any 512K blocks available?
        ii. No. Allocate a 1MB block.
            i. Are there any 1M blocks available?
            ii. Yes. Remove it from the list.
        iii. Split into two 512K blocks. Add one to the 512K list.
        iv. Return the other 512K block.
    c. Split into two 256K blocks. Add one to the 256K list.
    d. Return the other 256K block.
4. Split into two 128K blocks. Add one to the 128K list.
5. Return the other 128K block.

Deallocating memory

When we deallocate a block, we add it to the appropriate list, then check to see if its 'buddy' (the other chip from off the same old block) is also on the list. If so, we can merge to two buddies, and add a block twice the size to the next higher list. Again, this recursively merges all buddies which can be merged. So assuming there is no other allocated memory in the system, and we wish to deallocate the block we allocated above:

1. Is the buddy of this block on the 128K list?
2. Yes. Remove the buddy from the 128K list.
3. Merge the two blocks and deallocate the merged (256K) block.
    a. Is the buddy of this 256K block on the 256K list?
    b. Yes. Remove the buddy from the 256K list.
    c. Merge the two blocks and deallocate the merged (512K) block.
        i. Is the buddy of this 512K block on the 512K list?
        ii. Yes. Remove the buddy from the 512K list.
        iii. Merge the two blocks and deallocate the merged (1M) block.
            i. Is the buddy of this 1M block on the 1M list?
            ii. No. Add this block to the 1M list

Advantage

Buddy system is faster. When a block of size 2k is freed, a hole of 2k memory size is searched to check if a merge is possible, whereas in other algorithms all the hole list must be searched.

Disadvantage

It is often become inefficient in terms of memory utilization. As all requests must be rounded up to a power of 2, a 35KB process is allocated to 64KB, thus wasting extra 29KB causing internal fragmentation. There may be holes between the buddies causing external fragmentation.

Program:

```
#include<stdio.h>
int tree[2049],i,j,k;
void segmentalloc(int,int),makedivided(int),makefree(int),printing(int,int);
int place(int),power(int,int);
void main()
{
        int totsize,cho,req;
        printf("\n***BUDDY SYSTEM REQUIREMENT***\n");
        printf("\n  Enter the Size of the memory  :  \n");
        scanf("%d",&totsize);   //takes input for total size
        do
        {
                printf("BUDDY SYSTEM \n");
                printf("  1)   Allocate 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  :  \n");
```

```
                scanf("%d",&cho);
                switch(cho)
                {
                        case 1:
                                printf("\nMEMORY ALLOCATION ");
                                printf("\n  Enter the Process size  : ");       //takes the
size of the process
                                scanf("%d",&req);
                                segmentalloc(totsize,req);          //allocates memory
                                break;
                        case 2:
                                printf("\nMEMORY DEALLOCATION ");
                                printf("\n Enter the process size  :  ");       //takes
process size
                                scanf("%d",&req);
                                makefree(req);          //deallocation
                                break;
                        case 3:
                                printf("\nMEMORY ALLOCATION MAP\n");
                                printing(totsize,0);            //memory map
                                break;
                        default:
                                printf("Invalid input");
                }
        }while(cho!=3);
}

void segmentalloc(int totsize,int request)
{
        int flevel=0,size;
        size=totsize;               //complete block
        if(request>totsize)                 //request is greater
        {
                printf("\n The system don't have enough free memory");
                return;
```

```
        }
        while(1)
        {
                if(request<size && request>(size/2))       //block decided
                        break;
                else
                {
                        size/=2;              //block divided
                        flevel++;             //number of divide
                }
        }
        for(i=power(2,flevel)-1;i<=(power(2,flevel+1)-2);i++)  //i=1 to 2
                if(tree[i]==0 && place(i))
                {
                        tree[i]=request;
                        makedivided(i);
                        printf("Result    :     Successful Allocation");
                        break;
                }
        if(i==power(2,flevel+1)-1)
        {

                printf(" The system don't have enough free memory");
                printf("  Suggession  :  Go for VIRTUAL Memory Mode");
        }
}

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;

}

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",tree[node]);
                else if(tree[node]==1)
                        printf("---> Divided");
                else printf("---> Free");

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

```
            printing(totsize,2*node+2);
        }

}
```

Output:

```
shaziya@shaziya:~$ cd 26
shaziya@shaziya:~$ cd 26
shaziya@shaziya:~/26$ gcc buddy.c
shaziya@shaziya:~/26$ ./a.out

***BUDDY SYSTEM REQUIREMENT***

  Enter the Size of the memory  :
128
BUDDY SYSTEM
  1)  Allocate the process into the Memory
  2)  Remove the process from Memory
  3)  Tree structure for Memory allocation Map
  4)  Exit
*  Enter your choice  :
1

MEMORY ALLOCATION
  Enter the Process size  : 18
Result    :     Successful AllocationBUDDY SYSTEM
  1)  Allocate the process into the Memory
  2)  Remove the process from Memory
  3)  Tree structure for Memory allocation Map
  4)  Exit
*  Enter your choice  :
1
```

```
MEMORY ALLOCATION
  Enter the Process size  : 40
Result    :     Successful AllocationBUDDY SYSTEM
  1)  Allocate the process into the Memory
  2)  Remove the process from Memory
  3)  Tree structure for Memory allocation Map
  4)  Exit
*  Enter your choice  :
1

MEMORY ALLOCATION
  Enter the Process size  : 17
Result    :     Successful AllocationBUDDY SYSTEM
  1)  Allocate the process into the Memory
  2)  Remove the process from Memory
  3)  Tree structure for Memory allocation Map
  4)  Exit
*  Enter your choice  :
3

MEMORY ALLOCATION MAP
 128 ---> Divided        64 ---> Divided              32 ---> Allocated 18             32
---> Allocated 17        64 ---> Allocated 40shaziya@shaziya:~/26$ gcc buddy.c
shaziya@shaziya:~/26$ gcc buddy.c
shaziya@shaziya:~/26$ ./a.out

***BUDDY SYSTEM REQUIREMENT***

  Enter the Size of the memory  :
128
BUDDY SYSTEM
  1)  Allocate the process into the Memory
  2)  Remove the process from Memory
  3)  Tree structure for Memory allocation Map
  4)  Exit
```

```
*  Enter your choice  :
1

MEMORY ALLOCATION
  Enter the Process size  : 18
Result    :     Successful AllocationBUDDY SYSTEM
  1)   Allocate the process into the Memory
  2)   Remove the process from Memory
  3)   Tree structure for Memory allocation Map
  4)   Exit
*  Enter your choice  :
1

MEMORY ALLOCATION
  Enter the Process size  : 40
Result    :     Successful AllocationBUDDY SYSTEM
  1)   Allocate the process into the Memory
  2)   Remove the process from Memory
  3)   Tree structure for Memory allocation Map
  4)   Exit
*  Enter your choice  :
2

MEMORY DEALLOCATION
 Enter the process size  :  40
BUDDY SYSTEM
  1)   Allocate the process into the Memory
  2)   Remove the process from Memory
  3)   Tree structure for Memory allocation Map
  4)   Exit
*  Enter your choice  :
3

MEMORY ALLOCATION MAP
```

```
 128 ---> Divided       64 ---> Divided            32 ---> Allocated 18           32
---> Free       64 ---> Freeshaziya@shaziya:~/26$ gcc buddy.c
shaziya@shaziya:~/26$ gcc buddy.c
shaziya@shaziya:~/26$ ./a.out

***BUDDY SYSTEM REQUIREMENT***

  Enter the Size of the memory  :
128
BUDDY SYSTEM
  1)  Allocate the process into the Memory
  2)  Remove the process from Memory
  3)  Tree structure for Memory allocation Map
  4)  Exit
*  Enter your choice  :
1

MEMORY ALLOCATION
  Enter the Process size  : 130

 The system don't have enough free memory.
```

Conclusion

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.

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.

Reference

www.wikipedia.org

www.geeksforgeek.org

www.scribd.com