Operating Systems - EECE.5730

Instructor: *Prof. Dalila Megherbi*

Assignment – 3

Due by 11-03-16

By,

- Naga Ganesh Kurapati

1) Objective:

The purpose of this Lab is to initialize MMS and User threads. User threads request the memory blocks and MMS thread manages the all the memory blocks to meet the memory requests. And MMS take care of freeing the memory block and situation where it runs out of memory blocks. The synchronization is achieved using one mutex and semaphores.

2) Background:

Mutex are used to synchronize the threads accessing the critical sections on the program. It has can be either 1 or 0. 1 if thread has lock to access the critical section where as 0 then it has to wait for the lock. Semaphore can allow multiple threads to access the critical section. It can have more than two values unlike mutex. If it is 0, thread has to wait for the semaphore, if it is >0 thread can take the semaphore to access the critical section. Also thread can post the value to the semaphore after it done with the critical section. Memory management unit manages the memory blocks requests in a system according to the algorithms first fit, best fit and worst fit. It also takes care of the situation where it run out of memory blocks, in such situation it combines the small memory blocks available to make a larger block. And also it kicks out the lower priority thread to allocate memory to higher priority thread.

3) Algorithms/Functions used:

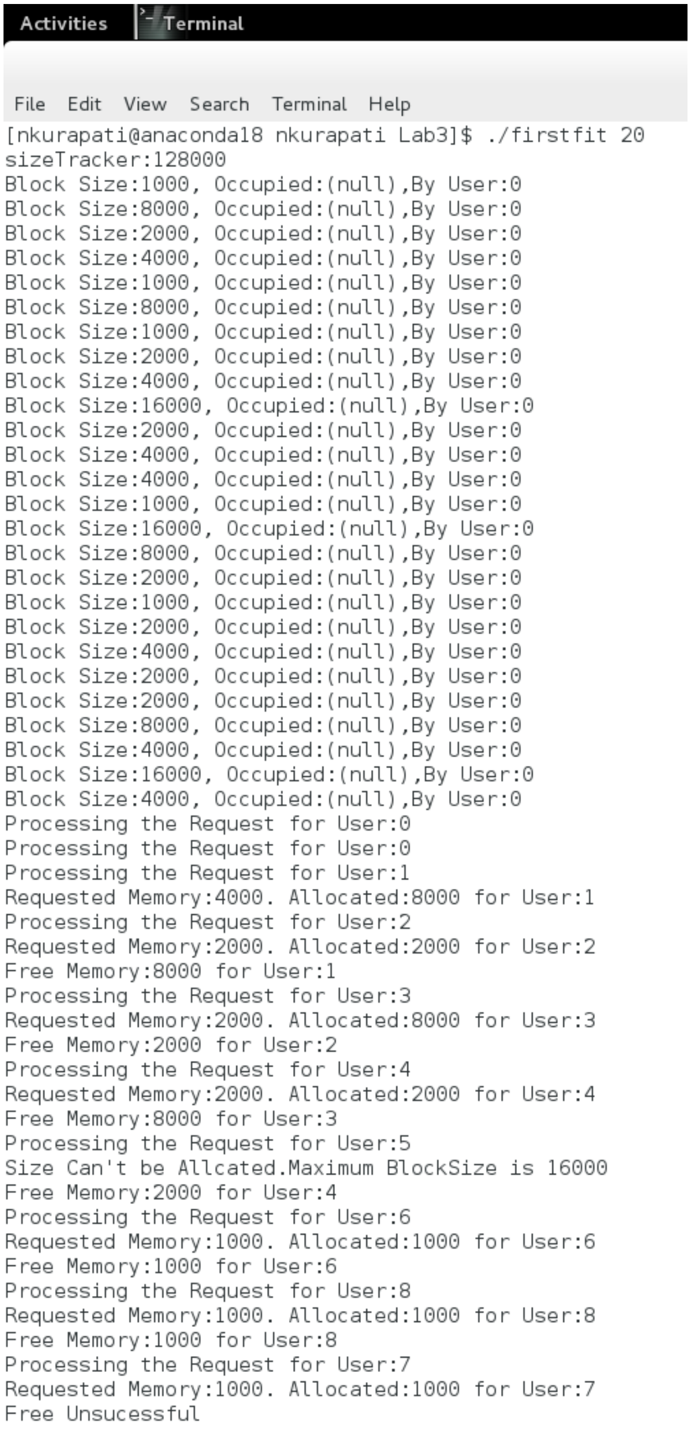
pthread_mutex_init() - to initialize mutex
sem_init() – to initialize semaphore
malloc() – to allocate memory
pthread_create() – to create pthreads
pthread_join() – to wait for threads to complete and join
pthread_mutex_destroy() – destroy mutex
sem_destroy() – destroy semaphore
sem_wait() – wait on semaphore
sem_post () – post the semaphore
pthread_mutex_lock() – mutex lock
pthread_mutex_unlock() -mutex unlock

Userdefined functions:

Manage() – to manage memory blocks
Request() – to request MMS to allocate or free memory block
memory_malloc() – to place the request on to the request buffer
Process_Request() – to read the memory block request by the user from the request buffer
memory_free() – to place the free memory block request on to the free buffer
Process_Free() – to read the memory free request by the user from the free buffer
Free_mBlock() – to free the memory block in the linked list
First_Fit() – to assign memory blocks using first fit algorithm
Best_Fit() – to assign memory blocks using best fit algorithm
Worst_Fit() – to assign memory blocks using worst fit algorithm

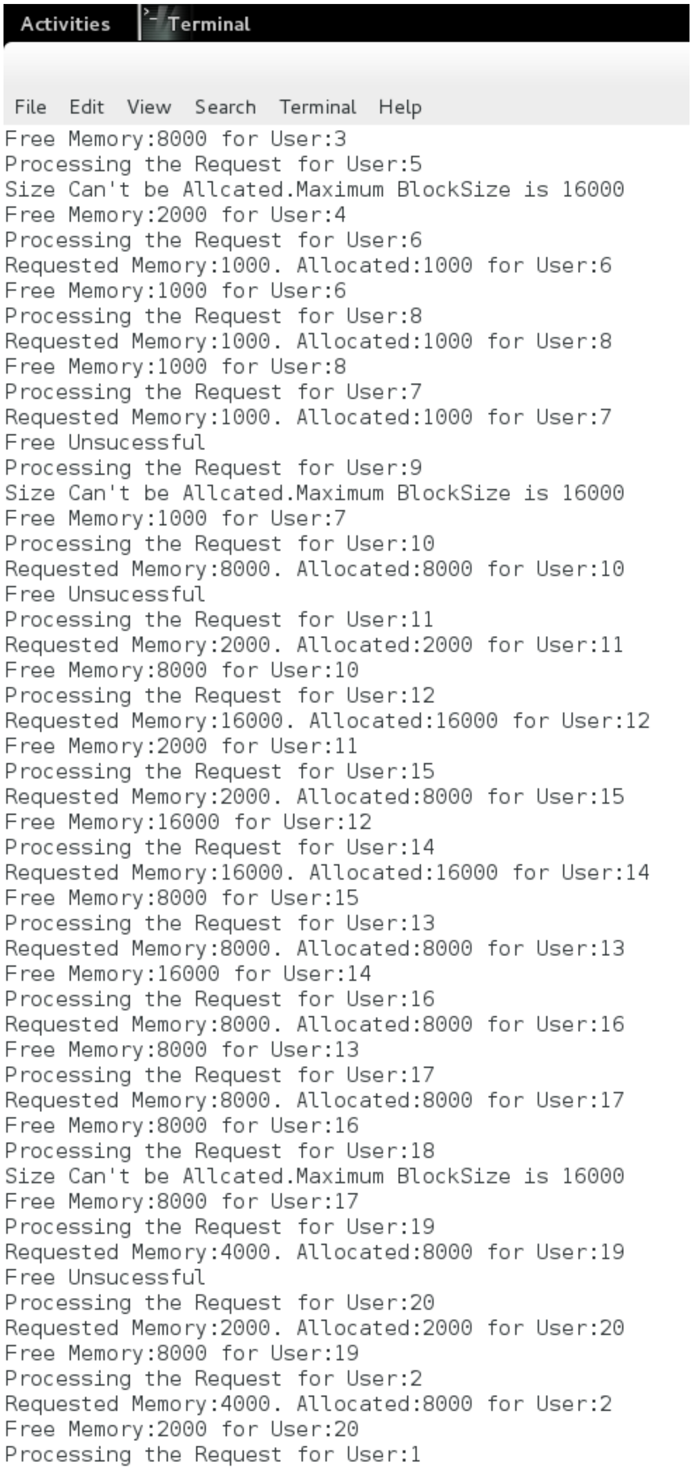
4) Results:

First fit with 20 user threads. You can see the memory blocks initialized during the start



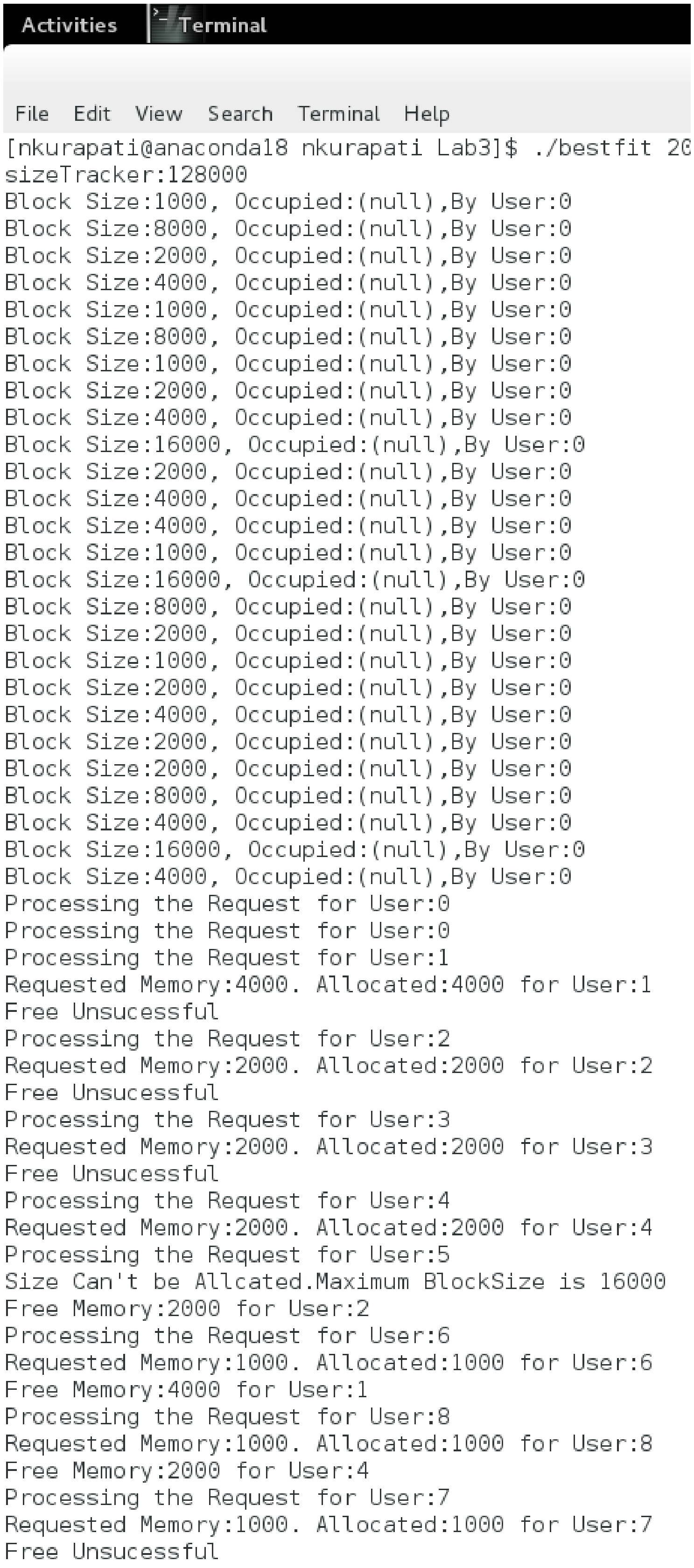
```
[nkurapati@anaconda18 nkurapati Lab3]$ ./firstfit 20
sizeTracker:128000
Block Size:1000, Occupied:(null),By User:0
Block Size:8000, Occupied:(null),By User:0
Block Size:2000, Occupied:(null),By User:0
Block Size:4000, Occupied:(null),By User:0
Block Size:1000, Occupied:(null),By User:0
Block Size:8000, Occupied:(null),By User:0
Block Size:1000, Occupied:(null),By User:0
Block Size:2000, Occupied:(null),By User:0
Block Size:4000, Occupied:(null),By User:0
Block Size:16000, Occupied:(null),By User:0
Block Size:2000, Occupied:(null),By User:0
Block Size:4000, Occupied:(null),By User:0
Block Size:4000, Occupied:(null),By User:0
Block Size:1000, Occupied:(null),By User:0
Block Size:16000, Occupied:(null),By User:0
Block Size:8000, Occupied:(null),By User:0
Block Size:2000, Occupied:(null),By User:0
Block Size:1000, Occupied:(null),By User:0
Block Size:2000, Occupied:(null),By User:0
Block Size:4000, Occupied:(null),By User:0
Block Size:2000, Occupied:(null),By User:0
Block Size:2000, Occupied:(null),By User:0
Block Size:8000, Occupied:(null),By User:0
Block Size:4000, Occupied:(null),By User:0
Block Size:16000, Occupied:(null),By User:0
Block Size:4000, Occupied:(null),By User:0
Processing the Request for User:0
Processing the Request for User:0
Processing the Request for User:1
Requested Memory:4000. Allocated:8000 for User:1
Processing the Request for User:2
Requested Memory:2000. Allocated:2000 for User:2
Free Memory:8000 for User:1
Processing the Request for User:3
Requested Memory:2000. Allocated:8000 for User:3
Free Memory:2000 for User:2
Processing the Request for User:4
Requested Memory:2000. Allocated:2000 for User:4
Free Memory:8000 for User:3
Processing the Request for User:5
Size Can't be Allcated.Maximum BlockSize is 16000
Free Memory:2000 for User:4
Processing the Request for User:6
Requested Memory:1000. Allocated:1000 for User:6
Free Memory:1000 for User:6
Processing the Request for User:8
Requested Memory:1000. Allocated:1000 for User:8
Free Memory:1000 for User:8
Processing the Request for User:7
Requested Memory:1000. Allocated:1000 for User:7
Free Unsucessful
```

First fit: Below you can see the memory blocks assigned accordingly. Defragment is not implemented in this case.



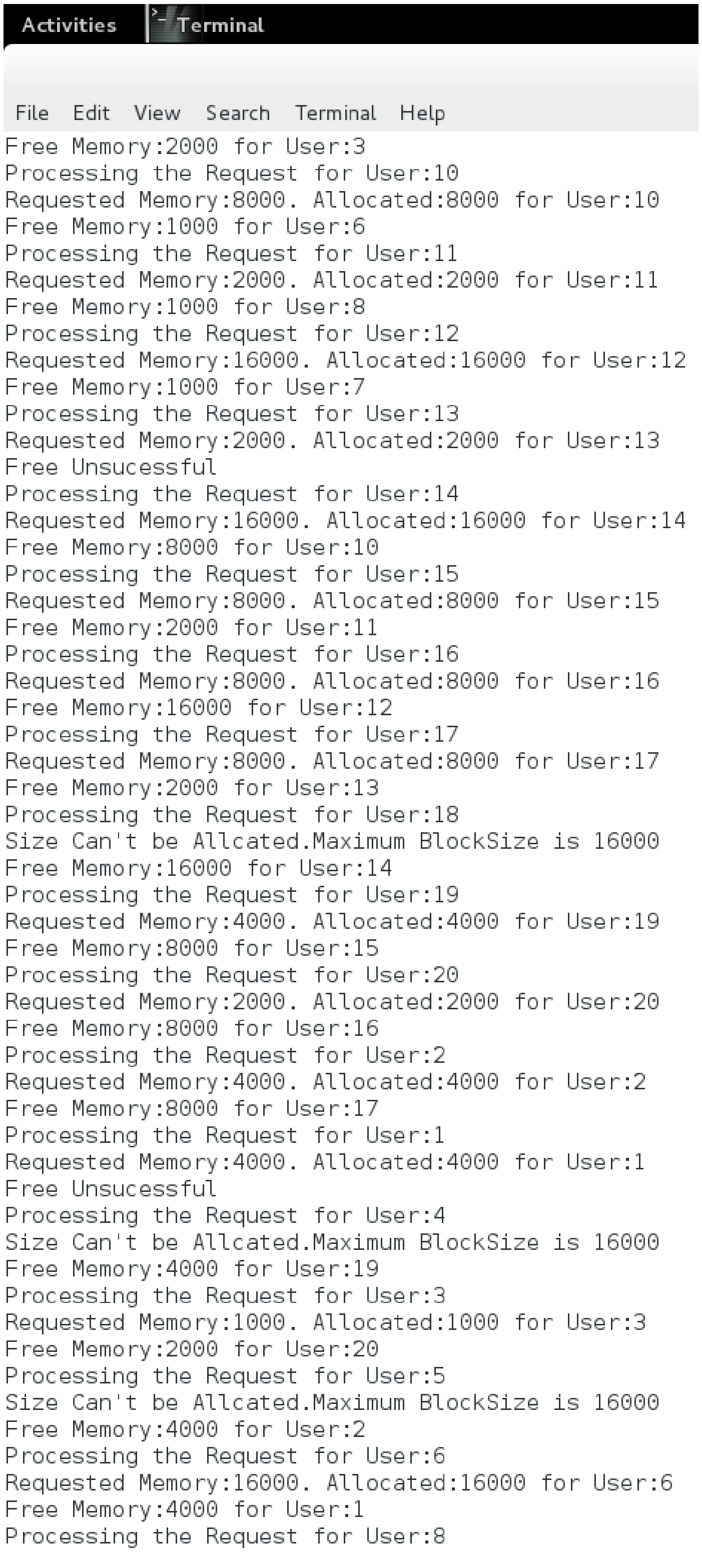
```
Free Memory:8000 for User:3
Processing the Request for User:5
Size Can't be Allcated.Maximum BlockSize is 16000
Free Memory:2000 for User:4
Processing the Request for User:6
Requested Memory:1000. Allocated:1000 for User:6
Free Memory:1000 for User:6
Processing the Request for User:8
Requested Memory:1000. Allocated:1000 for User:8
Free Memory:1000 for User:8
Processing the Request for User:7
Requested Memory:1000. Allocated:1000 for User:7
Free Unsucessful
Processing the Request for User:9
Size Can't be Allcated.Maximum BlockSize is 16000
Free Memory:1000 for User:7
Processing the Request for User:10
Requested Memory:8000. Allocated:8000 for User:10
Free Unsucessful
Processing the Request for User:11
Requested Memory:2000. Allocated:2000 for User:11
Free Memory:8000 for User:10
Processing the Request for User:12
Requested Memory:16000. Allocated:16000 for User:12
Free Memory:2000 for User:11
Processing the Request for User:15
Requested Memory:2000. Allocated:8000 for User:15
Free Memory:16000 for User:12
Processing the Request for User:14
Requested Memory:16000. Allocated:16000 for User:14
Free Memory:8000 for User:15
Processing the Request for User:13
Requested Memory:8000. Allocated:8000 for User:13
Free Memory:16000 for User:14
Processing the Request for User:16
Requested Memory:8000. Allocated:8000 for User:16
Free Memory:8000 for User:13
Processing the Request for User:17
Requested Memory:8000. Allocated:8000 for User:17
Free Memory:8000 for User:16
Processing the Request for User:18
Size Can't be Allcated.Maximum BlockSize is 16000
Free Memory:8000 for User:17
Processing the Request for User:19
Requested Memory:4000. Allocated:8000 for User:19
Free Unsucessful
Processing the Request for User:20
Requested Memory:2000. Allocated:2000 for User:20
Free Memory:8000 for User:19
Processing the Request for User:2
Requested Memory:4000. Allocated:8000 for User:2
Free Memory:2000 for User:20
Processing the Request for User:1
```

Best fit with 20 user threads. You can see the memory blocks initialized during the start



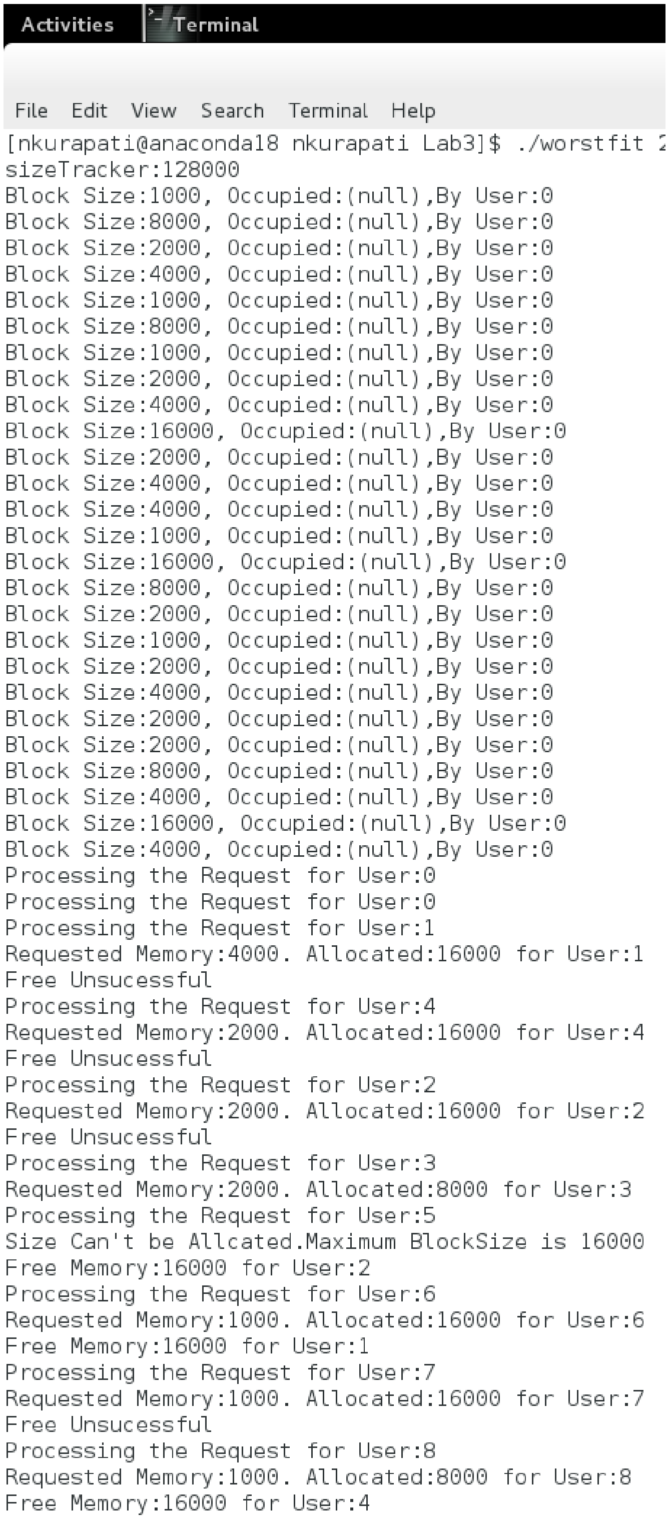
```
[nkurapati@anaconda18 nkurapati Lab3]$ ./bestfit 20
sizeTracker:128000
Block Size:1000, Occupied:(null),By User:0
Block Size:8000, Occupied:(null),By User:0
Block Size:2000, Occupied:(null),By User:0
Block Size:4000, Occupied:(null),By User:0
Block Size:1000, Occupied:(null),By User:0
Block Size:8000, Occupied:(null),By User:0
Block Size:1000, Occupied:(null),By User:0
Block Size:2000, Occupied:(null),By User:0
Block Size:4000, Occupied:(null),By User:0
Block Size:16000, Occupied:(null),By User:0
Block Size:2000, Occupied:(null),By User:0
Block Size:4000, Occupied:(null),By User:0
Block Size:4000, Occupied:(null),By User:0
Block Size:1000, Occupied:(null),By User:0
Block Size:16000, Occupied:(null),By User:0
Block Size:8000, Occupied:(null),By User:0
Block Size:2000, Occupied:(null),By User:0
Block Size:1000, Occupied:(null),By User:0
Block Size:2000, Occupied:(null),By User:0
Block Size:4000, Occupied:(null),By User:0
Block Size:2000, Occupied:(null),By User:0
Block Size:2000, Occupied:(null),By User:0
Block Size:8000, Occupied:(null),By User:0
Block Size:4000, Occupied:(null),By User:0
Block Size:16000, Occupied:(null),By User:0
Block Size:4000, Occupied:(null),By User:0
Processing the Request for User:0
Processing the Request for User:0
Processing the Request for User:1
Requested Memory:4000. Allocated:4000 for User:1
Free Unsucessful
Processing the Request for User:2
Requested Memory:2000. Allocated:2000 for User:2
Free Unsucessful
Processing the Request for User:3
Requested Memory:2000. Allocated:2000 for User:3
Free Unsucessful
Processing the Request for User:4
Requested Memory:2000. Allocated:2000 for User:4
Processing the Request for User:5
Size Can't be Allcated.Maximum BlockSize is 16000
Free Memory:2000 for User:2
Processing the Request for User:6
Requested Memory:1000. Allocated:1000 for User:6
Free Memory:4000 for User:1
Processing the Request for User:8
Requested Memory:1000. Allocated:1000 for User:8
Free Memory:2000 for User:4
Processing the Request for User:7
Requested Memory:1000. Allocated:1000 for User:7
Free Unsucessful
```

Best fit: Below you can see the memory blocks assigned accordingly. Defragment is not implemented in this case.



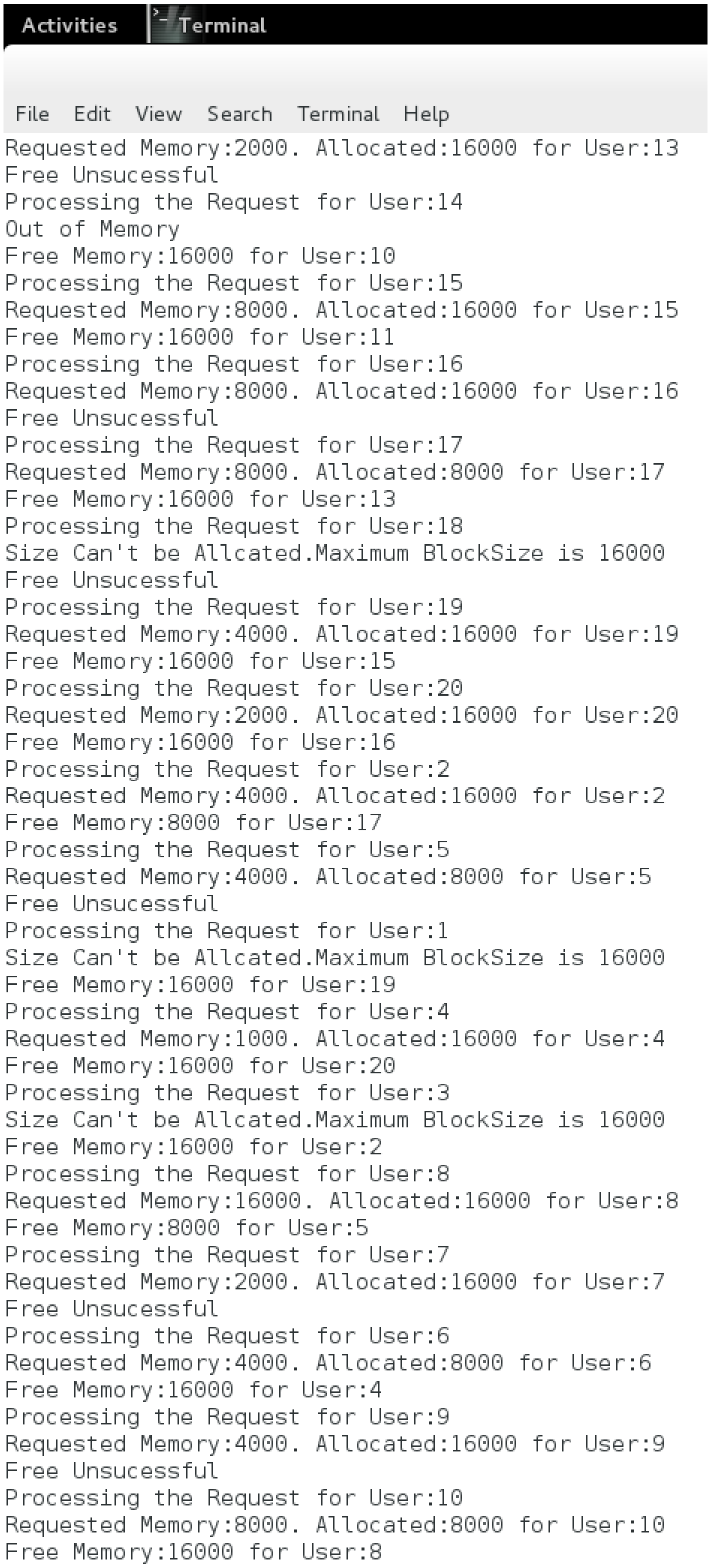
```
Free Memory:2000 for User:3
Processing the Request for User:10
Requested Memory:8000. Allocated:8000 for User:10
Free Memory:1000 for User:6
Processing the Request for User:11
Requested Memory:2000. Allocated:2000 for User:11
Free Memory:1000 for User:8
Processing the Request for User:12
Requested Memory:16000. Allocated:16000 for User:12
Free Memory:1000 for User:7
Processing the Request for User:13
Requested Memory:2000. Allocated:2000 for User:13
Free Unsucessful
Processing the Request for User:14
Requested Memory:16000. Allocated:16000 for User:14
Free Memory:8000 for User:10
Processing the Request for User:15
Requested Memory:8000. Allocated:8000 for User:15
Free Memory:2000 for User:11
Processing the Request for User:16
Requested Memory:8000. Allocated:8000 for User:16
Free Memory:16000 for User:12
Processing the Request for User:17
Requested Memory:8000. Allocated:8000 for User:17
Free Memory:2000 for User:13
Processing the Request for User:18
Size Can't be Allcated.Maximum BlockSize is 16000
Free Memory:16000 for User:14
Processing the Request for User:19
Requested Memory:4000. Allocated:4000 for User:19
Free Memory:8000 for User:15
Processing the Request for User:20
Requested Memory:2000. Allocated:2000 for User:20
Free Memory:8000 for User:16
Processing the Request for User:2
Requested Memory:4000. Allocated:4000 for User:2
Free Memory:8000 for User:17
Processing the Request for User:1
Requested Memory:4000. Allocated:4000 for User:1
Free Unsucessful
Processing the Request for User:4
Size Can't be Allcated.Maximum BlockSize is 16000
Free Memory:4000 for User:19
Processing the Request for User:3
Requested Memory:1000. Allocated:1000 for User:3
Free Memory:2000 for User:20
Processing the Request for User:5
Size Can't be Allcated.Maximum BlockSize is 16000
Free Memory:4000 for User:2
Processing the Request for User:6
Requested Memory:16000. Allocated:16000 for User:6
Free Memory:4000 for User:1
Processing the Request for User:8
```

Worst fit with 20 user threads. You can see the memory blocks initialized during the start.



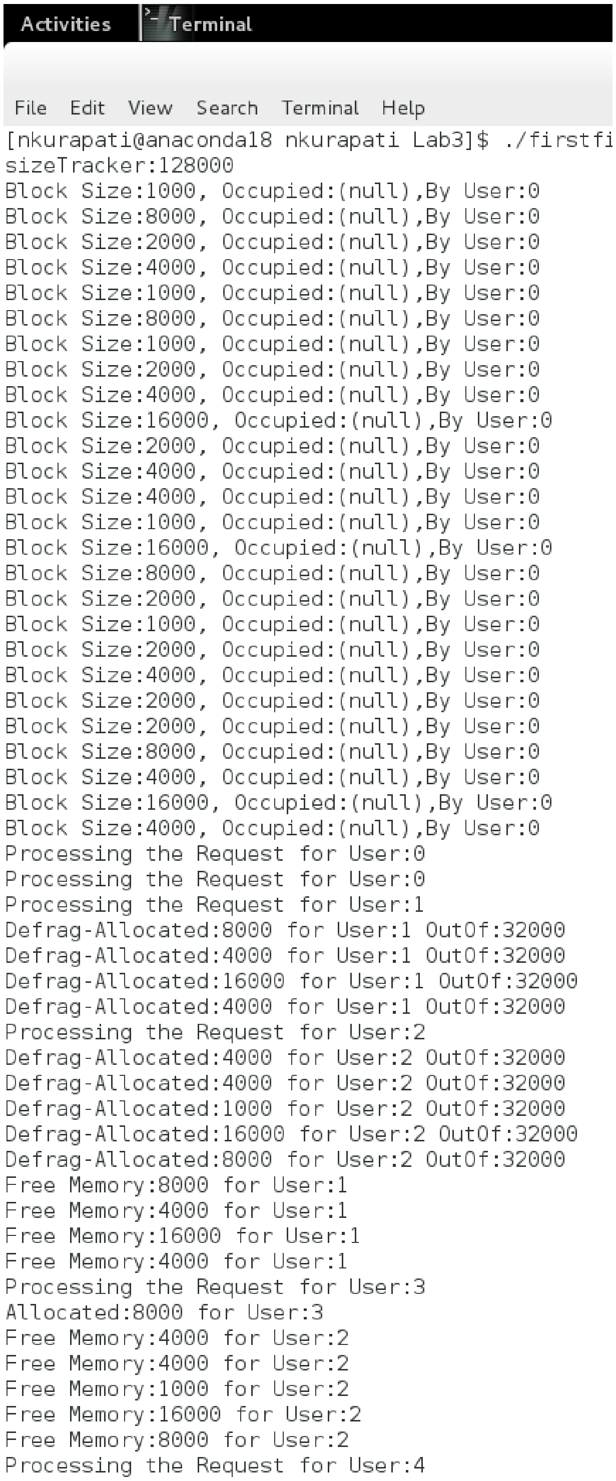
```
[nkurapati@anaconda18 nkurapati Lab3]$ ./worstfit 2
sizeTracker:128000
Block Size:1000, Occupied:(null),By User:0
Block Size:8000, Occupied:(null),By User:0
Block Size:2000, Occupied:(null),By User:0
Block Size:4000, Occupied:(null),By User:0
Block Size:1000, Occupied:(null),By User:0
Block Size:8000, Occupied:(null),By User:0
Block Size:1000, Occupied:(null),By User:0
Block Size:2000, Occupied:(null),By User:0
Block Size:4000, Occupied:(null),By User:0
Block Size:16000, Occupied:(null),By User:0
Block Size:2000, Occupied:(null),By User:0
Block Size:4000, Occupied:(null),By User:0
Block Size:4000, Occupied:(null),By User:0
Block Size:1000, Occupied:(null),By User:0
Block Size:16000, Occupied:(null),By User:0
Block Size:8000, Occupied:(null),By User:0
Block Size:2000, Occupied:(null),By User:0
Block Size:1000, Occupied:(null),By User:0
Block Size:2000, Occupied:(null),By User:0
Block Size:4000, Occupied:(null),By User:0
Block Size:2000, Occupied:(null),By User:0
Block Size:2000, Occupied:(null),By User:0
Block Size:8000, Occupied:(null),By User:0
Block Size:4000, Occupied:(null),By User:0
Block Size:16000, Occupied:(null),By User:0
Block Size:4000, Occupied:(null),By User:0
Processing the Request for User:0
Processing the Request for User:0
Processing the Request for User:1
Requested Memory:4000. Allocated:16000 for User:1
Free Unsucessful
Processing the Request for User:4
Requested Memory:2000. Allocated:16000 for User:4
Free Unsucessful
Processing the Request for User:2
Requested Memory:2000. Allocated:16000 for User:2
Free Unsucessful
Processing the Request for User:3
Requested Memory:2000. Allocated:8000 for User:3
Processing the Request for User:5
Size Can't be Allcated.Maximum BlockSize is 16000
Free Memory:16000 for User:2
Processing the Request for User:6
Requested Memory:1000. Allocated:16000 for User:6
Free Memory:16000 for User:1
Processing the Request for User:7
Requested Memory:1000. Allocated:16000 for User:7
Free Unsucessful
Processing the Request for User:8
Requested Memory:1000. Allocated:8000 for User:8
Free Memory:16000 for User:4
```

Worst fit: Below you can see the memory blocks assigned accordingly. Defragment is not implemented in this case.



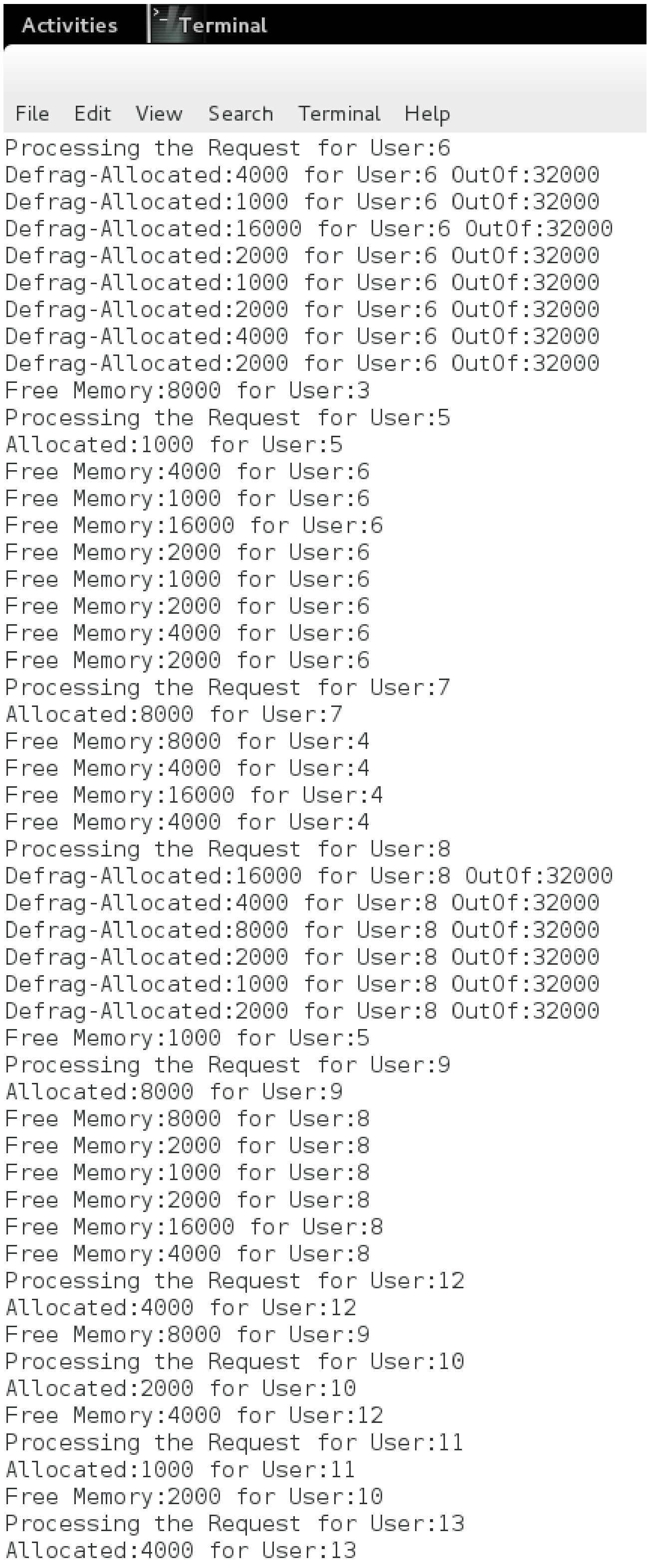
```
Requested Memory:2000. Allocated:16000 for User:13
Free Unsucessful
Processing the Request for User:14
Out of Memory
Free Memory:16000 for User:10
Processing the Request for User:15
Requested Memory:8000. Allocated:16000 for User:15
Free Memory:16000 for User:11
Processing the Request for User:16
Requested Memory:8000. Allocated:16000 for User:16
Free Unsucessful
Processing the Request for User:17
Requested Memory:8000. Allocated:8000 for User:17
Free Memory:16000 for User:13
Processing the Request for User:18
Size Can't be Allcated.Maximum BlockSize is 16000
Free Unsucessful
Processing the Request for User:19
Requested Memory:4000. Allocated:16000 for User:19
Free Memory:16000 for User:15
Processing the Request for User:20
Requested Memory:2000. Allocated:16000 for User:20
Free Memory:16000 for User:16
Processing the Request for User:2
Requested Memory:4000. Allocated:16000 for User:2
Free Memory:8000 for User:17
Processing the Request for User:5
Requested Memory:4000. Allocated:8000 for User:5
Free Unsucessful
Processing the Request for User:1
Size Can't be Allcated.Maximum BlockSize is 16000
Free Memory:16000 for User:19
Processing the Request for User:4
Requested Memory:1000. Allocated:16000 for User:4
Free Memory:16000 for User:20
Processing the Request for User:3
Size Can't be Allcated.Maximum BlockSize is 16000
Free Memory:16000 for User:2
Processing the Request for User:8
Requested Memory:16000. Allocated:16000 for User:8
Free Memory:8000 for User:5
Processing the Request for User:7
Requested Memory:2000. Allocated:16000 for User:7
Free Unsucessful
Processing the Request for User:6
Requested Memory:4000. Allocated:8000 for User:6
Free Memory:16000 for User:4
Processing the Request for User:9
Requested Memory:4000. Allocated:16000 for User:9
Free Unsucessful
Processing the Request for User:10
Requested Memory:8000. Allocated:8000 for User:10
Free Memory:16000 for User:8
```

First fit: Below you can see the memory blocks assigned accordingly. Defragment is implemented in this case. See the following three figs.



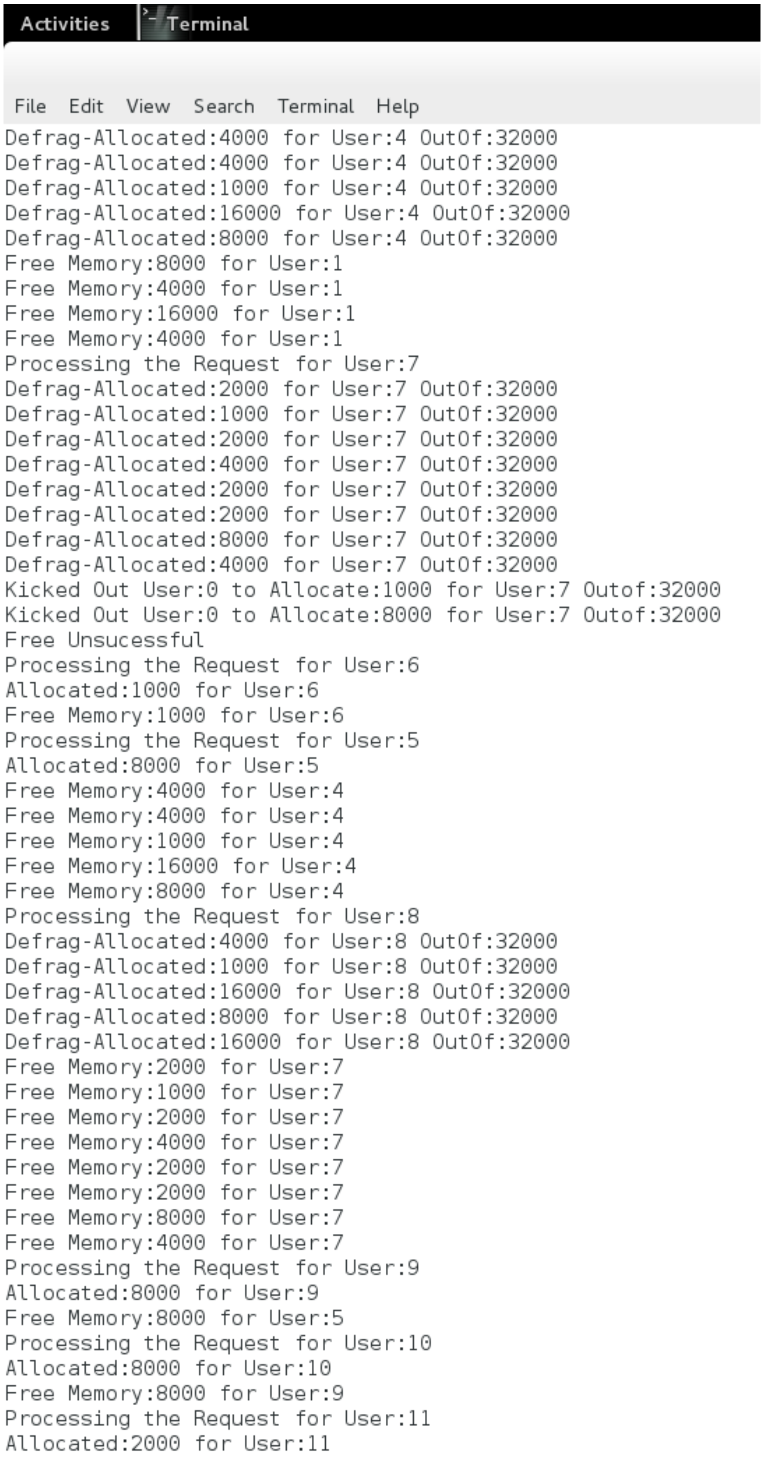
```
[nkurapati@anaconda18 nkurapati Lab3]$ ./firstfi
sizeTracker:128000
Block Size:1000, Occupied:(null),By User:0
Block Size:8000, Occupied:(null),By User:0
Block Size:2000, Occupied:(null),By User:0
Block Size:4000, Occupied:(null),By User:0
Block Size:1000, Occupied:(null),By User:0
Block Size:8000, Occupied:(null),By User:0
Block Size:1000, Occupied:(null),By User:0
Block Size:2000, Occupied:(null),By User:0
Block Size:4000, Occupied:(null),By User:0
Block Size:16000, Occupied:(null),By User:0
Block Size:2000, Occupied:(null),By User:0
Block Size:4000, Occupied:(null),By User:0
Block Size:4000, Occupied:(null),By User:0
Block Size:1000, Occupied:(null),By User:0
Block Size:16000, Occupied:(null),By User:0
Block Size:8000, Occupied:(null),By User:0
Block Size:2000, Occupied:(null),By User:0
Block Size:1000, Occupied:(null),By User:0
Block Size:2000, Occupied:(null),By User:0
Block Size:4000, Occupied:(null),By User:0
Block Size:2000, Occupied:(null),By User:0
Block Size:2000, Occupied:(null),By User:0
Block Size:8000, Occupied:(null),By User:0
Block Size:4000, Occupied:(null),By User:0
Block Size:16000, Occupied:(null),By User:0
Block Size:4000, Occupied:(null),By User:0
Processing the Request for User:0
Processing the Request for User:0
Processing the Request for User:1
Defrag-Allocated:8000 for User:1 OutOf:32000
Defrag-Allocated:4000 for User:1 OutOf:32000
Defrag-Allocated:16000 for User:1 OutOf:32000
Defrag-Allocated:4000 for User:1 OutOf:32000
Processing the Request for User:2
Defrag-Allocated:4000 for User:2 OutOf:32000
Defrag-Allocated:4000 for User:2 OutOf:32000
Defrag-Allocated:1000 for User:2 OutOf:32000
Defrag-Allocated:16000 for User:2 OutOf:32000
Defrag-Allocated:8000 for User:2 OutOf:32000
Free Memory:8000 for User:1
Free Memory:4000 for User:1
Free Memory:16000 for User:1
Free Memory:4000 for User:1
Processing the Request for User:3
Allocated:8000 for User:3
Free Memory:4000 for User:2
Free Memory:4000 for User:2
Free Memory:1000 for User:2
Free Memory:16000 for User:2
Free Memory:8000 for User:2
Processing the Request for User:4
```

First fit: Below you can see the memory blocks assigned accordingly. Defragment is implemented in this case.



```
Processing the Request for User:6
Defrag-Allocated:4000 for User:6 OutOf:32000
Defrag-Allocated:1000 for User:6 OutOf:32000
Defrag-Allocated:16000 for User:6 OutOf:32000
Defrag-Allocated:2000 for User:6 OutOf:32000
Defrag-Allocated:1000 for User:6 OutOf:32000
Defrag-Allocated:2000 for User:6 OutOf:32000
Defrag-Allocated:4000 for User:6 OutOf:32000
Defrag-Allocated:2000 for User:6 OutOf:32000
Free Memory:8000 for User:3
Processing the Request for User:5
Allocated:1000 for User:5
Free Memory:4000 for User:6
Free Memory:1000 for User:6
Free Memory:16000 for User:6
Free Memory:2000 for User:6
Free Memory:1000 for User:6
Free Memory:2000 for User:6
Free Memory:4000 for User:6
Free Memory:2000 for User:6
Processing the Request for User:7
Allocated:8000 for User:7
Free Memory:8000 for User:4
Free Memory:4000 for User:4
Free Memory:16000 for User:4
Free Memory:4000 for User:4
Processing the Request for User:8
Defrag-Allocated:16000 for User:8 OutOf:32000
Defrag-Allocated:4000 for User:8 OutOf:32000
Defrag-Allocated:8000 for User:8 OutOf:32000
Defrag-Allocated:2000 for User:8 OutOf:32000
Defrag-Allocated:1000 for User:8 OutOf:32000
Defrag-Allocated:2000 for User:8 OutOf:32000
Free Memory:1000 for User:5
Processing the Request for User:9
Allocated:8000 for User:9
Free Memory:8000 for User:8
Free Memory:2000 for User:8
Free Memory:1000 for User:8
Free Memory:2000 for User:8
Free Memory:16000 for User:8
Free Memory:4000 for User:8
Processing the Request for User:12
Allocated:4000 for User:12
Free Memory:8000 for User:9
Processing the Request for User:10
Allocated:2000 for User:10
Free Memory:4000 for User:12
Processing the Request for User:11
Allocated:1000 for User:11
Free Memory:2000 for User:10
Processing the Request for User:13
Allocated:4000 for User:13
```

First fit: Below you can see the memory blocks assigned accordingly. Defragment is implemented in this case. You can see the kicking out the lower priority thread to get free memory block.



```
Defrag-Allocated:4000 for User:4 OutOf:32000
Defrag-Allocated:4000 for User:4 OutOf:32000
Defrag-Allocated:1000 for User:4 OutOf:32000
Defrag-Allocated:16000 for User:4 OutOf:32000
Defrag-Allocated:8000 for User:4 OutOf:32000
Free Memory:8000 for User:1
Free Memory:4000 for User:1
Free Memory:16000 for User:1
Free Memory:4000 for User:1
Processing the Request for User:7
Defrag-Allocated:2000 for User:7 OutOf:32000
Defrag-Allocated:1000 for User:7 OutOf:32000
Defrag-Allocated:2000 for User:7 OutOf:32000
Defrag-Allocated:4000 for User:7 OutOf:32000
Defrag-Allocated:2000 for User:7 OutOf:32000
Defrag-Allocated:2000 for User:7 OutOf:32000
Defrag-Allocated:8000 for User:7 OutOf:32000
Defrag-Allocated:4000 for User:7 OutOf:32000
Kicked Out User:0 to Allocate:1000 for User:7 Outof:32000
Kicked Out User:0 to Allocate:8000 for User:7 Outof:32000
Free Unsucessful
Processing the Request for User:6
Allocated:1000 for User:6
Free Memory:1000 for User:6
Processing the Request for User:5
Allocated:8000 for User:5
Free Memory:4000 for User:4
Free Memory:4000 for User:4
Free Memory:1000 for User:4
Free Memory:16000 for User:4
Free Memory:8000 for User:4
Processing the Request for User:8
Defrag-Allocated:4000 for User:8 OutOf:32000
Defrag-Allocated:1000 for User:8 OutOf:32000
Defrag-Allocated:16000 for User:8 OutOf:32000
Defrag-Allocated:8000 for User:8 OutOf:32000
Defrag-Allocated:16000 for User:8 OutOf:32000
Free Memory:2000 for User:7
Free Memory:1000 for User:7
Free Memory:2000 for User:7
Free Memory:4000 for User:7
Free Memory:2000 for User:7
Free Memory:2000 for User:7
Free Memory:8000 for User:7
Free Memory:4000 for User:7
Processing the Request for User:9
Allocated:8000 for User:9
Free Memory:8000 for User:5
Processing the Request for User:10
Allocated:8000 for User:10
Free Memory:8000 for User:9
Processing the Request for User:11
Allocated:2000 for User:11
```

5) Observations:

Buffers are very useful in managing the communication between the threads. And Linked list best useful in this case to manage the memory blocks.

6) Conclusions:

Successfully created the MMS and user threads according to the problem designed. MMS thread is able to manage the memory blocks effectively.

7) Source Code:

See the attachment to find the source code of first fit, best fit, worst fit and first fit with defragmentation respectively.

```c
#include<stdio.h>
#include<stdbool.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
#include<semaphore.h>
#include<signal.h>

//memory for MMS to manage in Bytes
#define memorySize 128000
#define MaxBlockSize 16000
//Define the size of the queue
#define BufferSize 10

//user thread to select a random size
int basket[6] = {1000,2000,4000,8000,16000,32000};
//memory pointers
void *mPtr;

//Mutex & semaphore
pthread_mutex_t mutex_req;
pthread_mutex_t mutex_free;
sem_t threadSem;
sem_t ReqSem;
sem_t FreeSem;

//Defintion of Memory Block
typedef struct mBlockStruct{
  int size; //size of the memory block
  bool occupied; //buffer to place items produced
  long threadIdOccupied;
  struct mBlockStruct *nextBlock; //pointer to next block
}mBlock;

//first memory block intialization
mBlock *firstBlock = NULL;

//size allocated over all
int sizeTracker = 0;

void pushOnMBlock(int sizeTmp)
{
   mBlock *current = firstBlock;
   while(current->nextBlock!=NULL)
   {
     current = current->nextBlock;
   }
   current->nextBlock = (mBlock *)malloc(sizeof(mBlock));
   current->nextBlock->size=sizeTmp;
   current->nextBlock->nextBlock=NULL;
}

void printMBlock()
{
   mBlock *current = firstBlock;
   while(current->nextBlock!=NULL)
   {
     printf("Block Size:%d, Occupied:%s,By User:%d\n",current->size,current->occupied,current-
>threadIdOccupied);
     current = current->nextBlock;
   }
}

//Defintion of Request Queue
typedef struct {
  int in;  //Number of Requests on queue
  int out; //Number of Requests taken
  long threadIdRequested[BufferSize]; //buffer to place threadID
  int sizeRequested[BufferSize]; //buffer to place size
}ReqQue;

//Intialize Request queue
ReqQue ReqQueBuf = {0,0,{0},{0}};

//Defintion of Free Queue
typedef struct {
  int in;  //Number of Requests on queue
  int out; //Number of Requests taken
  long threadIdRequested[BufferSize]; //buffer to place threadID
}FreeQue;

//Intialize Free queue
FreeQue FreeQueBuf = {0,0,{0}};

//To allocate the memory block
void First_Fit(int sizeReq, long threadIdReq)
{
   if((sizeReq==0)||(threadIdReq==0))
   {
     return;
   }
   if(sizeReq>MaxBlockSize)
   {
     printf("Size Can't be Allcated.Maximum BlockSize is %d\n",MaxBlockSize);
     return;
   }
   mBlock *current = firstBlock;
   while(current->nextBlock!=NULL)
   {
     if(((current->size)>=sizeReq) && ((current->occupied) == false))
     {
       current->occupied = true;
       current->threadIdOccupied = threadIdReq;
       printf("Allocated:%d for User:%d \n",current->size,threadIdReq);
       return;
     }
     current = current->nextBlock;
   }
   printf("Out of Memory\n");
}

//To free the memory block
void Free_mBlock(long threadIdReq)
{
  if(threadIdReq==0)
   {
     return;
   }
   mBlock *current = firstBlock;
   while(current->nextBlock!=NULL)
   {
     if((current->threadIdOccupied)==threadIdReq)
     {
       current->occupied = false;
       current->threadIdOccupied = 0;
       printf("Free Memory:%d for User:%d \n",current->size,threadIdReq);
       return;
     }
     current = current->nextBlock;
   }
   printf("Free Unsucessful\n");
}

//Memory requested by user
```

```
void memory_malloc(int sizeReq, long threadIdReq)
{
      //wait on request semephore
      sem_wait(&ReqSem);
      //wait for the critical area access
      pthread_mutex_lock(&mutex_req);
      //check if request queue is full
      if(!(((ReqQueBuf.in + 1) % BufferSize ) == ReqQueBuf.out))
        {
          //Place the request on buffer
          ReqQueBuf.threadIdRequested[ReqQueBuf.in] = threadIdReq;
          ReqQueBuf.sizeRequested[ReqQueBuf.in] = sizeReq;
          //printf("Size Requested:%d by User:%d \n", ReqQueBuf.sizeRequested
[ReqQueBuf.in],ReqQueBuf.threadIdRequested[ReqQueBuf.in]);
          //Increment the 'in' index
          ReqQueBuf.in = (ReqQueBuf.in + 1) % BufferSize;
        }
      else
        {
          printf("Queue is full when accessed by User:%d \n",threadIdReq);
        }
      //release the critical area access
      pthread_mutex_unlock(&mutex_req);
      sleep(1);
}

//Memory manager to process the memory allocation request
void Process_Request()
{
      int sizeReqt;
      long threadIdReqt;
      //wait for the critical area access
      pthread_mutex_lock(&mutex_req);
      //Check if request buffer is empty
      if(!(ReqQueBuf.in == ReqQueBuf.out))
        {
          printf("Processing the Request for User:%d \n",ReqQueBuf.threadIdRequested[ReqQueBuf.in]);
          //Consume the request
          sizeReqt = ReqQueBuf.sizeRequested[ReqQueBuf.in];
          threadIdReqt = ReqQueBuf.threadIdRequested[ReqQueBuf.in];
          //Increment the 'out' index
          ReqQueBuf.out = (ReqQueBuf.out + 1) % BufferSize;
        }
      //release the critical area access
      pthread_mutex_unlock(&mutex_req);
      //Post the request sem
      sem_post(&ReqSem);
      //Call the algorithm
      First_Fit(sizeReqt,threadIdReqt);
      sleep(1);
}

//Memory free request by user
void memory_free(long threadIdFree)
{
        //wait on free semephore
      sem_wait(&FreeSem);
      //wait for the critical area access
      pthread_mutex_lock(&mutex_free);
      //check if request queue is full
      if(!(((FreeQueBuf.in + 1) % BufferSize ) == FreeQueBuf.out))
        {
          //Place the request on buffer
          FreeQueBuf.threadIdRequested[FreeQueBuf.in] = threadIdFree;
          //printf("Size Free Requested by User:%d \n", threadIdFree);
          //Increment the 'in' index
          FreeQueBuf.in = (FreeQueBuf.in + 1) % BufferSize;
        }
      else
        {
          printf("Queue is full when accessed by User:%d \n", threadIdFree);
        }
      //release the critical area access
      pthread_mutex_unlock(&mutex_free);
      sleep(1);
}

//Memory Manager to process free request
void Process_Free()
{
      long threadIdReqt;
      //wait for the critical area access
      pthread_mutex_lock(&mutex_free);
      //Check if request buffer is empty
      if(!(FreeQueBuf.in == FreeQueBuf.out))
        {
          //printf("Processing the Free Request for User:%d \n",FreeQueBuf.threadIdRequested
[FreeQueBuf.in]);
          //Consume the request
          threadIdReqt = FreeQueBuf.threadIdRequested[FreeQueBuf.in];
          //Increment the 'out' index
          FreeQueBuf.out = (FreeQueBuf.out + 1) % BufferSize;
        }
      //release the critical area access
      pthread_mutex_unlock(&mutex_free);
      //Post the request sem
      sem_post(&FreeSem);
      Free_mBlock(threadIdReqt);
      sleep(1);
}

//Memory Manager
void *Manage(void* id)
{

        sleep(1);
        //printf("MMU:%d \n",(long)id);
        while(1)
        {
        Process_Request();
        Process_Free();
        }
}

//Request from user to allocate memory
void *Request(void* id)
{
        //printf("User:%d \n",(long)id);
        while(1)
        {
         int randNo = rand()%6;
        memory_malloc(basket[randNo],(long)id);
        sleep(4);
        memory_free((long)id);
        }
}


int main(int argc, char *argv[])
{
  //To get the number of users from cmd line
  unsigned int userNo = atoi(argv[1]);

  //intialize memory block
  firstBlock = (mBlock *)malloc(sizeof(mBlock));
  firstBlock->size = 1000;
  firstBlock->occupied = false;
  firstBlock->threadIdOccupied =0;
  firstBlock->nextBlock = NULL;
  sizeTracker = sizeTracker + 1000;
  //Allocate memory for the MMS
  mPtr = malloc(memorySize);
  //printf("mPtr:%d\n",mPtr);
  while(!(sizeTracker == memorySize))
  {
    int randNo = rand()%5;
    if(basket[randNo]<=(memorySize-sizeTracker))
    {
       pushOnMBlock(basket[randNo]);
       sizeTracker = sizeTracker+ basket[randNo];
    }
  }
  printf("sizeTracker:%d\n",sizeTracker);
  printMBlock();
  //Define MMU number
  unsigned int MMUNo = 1;
  pthread_t p[MMUNo];
  pthread_t* b;

  //Intialize mutex and two semaphores
  int e1 = pthread_mutex_init(&mutex_req, NULL);
  int e2 = pthread_mutex_init(&mutex_free, NULL);
  int e3 = sem_init(&ReqSem, 0, BufferSize-2);
  int e4 = sem_init(&FreeSem, 0, BufferSize-2);

  //Notify if failed
  if(e1!=0||e2!=0||e3!=0||e4!=0)
    printf("Intialization Error");

  b = malloc(userNo*sizeof(pthread_t));

  long i;
  //Create MMU threads
  for(i=0;i<MMUNo;i++)
    pthread_create(&p[i], NULL, Manage, (void*) i+1);

  //Create user threads
  for(i=0;i<userNo;i++)
    pthread_create(&b[i], NULL, Request, (void*) i+1);

  //Wait for MMU and user threads to finish
  for(i=0;i<MMUNo;i++)
    pthread_join(p[i], NULL);
  for(i=0;i<userNo;i++)
    pthread_join(b[i], NULL);

  //Destroy mutex and semaphores
  pthread_mutex_destroy(&mutex_req);
  pthread_mutex_destroy(&mutex_free);
  sem_destroy(&ReqSem);
  sem_destroy(&FreeSem);
  return 0;
}
```

```
#include<stdio.h>
#include<stdbool.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
#include<semaphore.h>
#include<signal.h>
#include<limits.h>

//memory for MMS to manage in Bytes
#define memorySize 128000
#define MaxBlockSize 16000
//Define the size of the queue
#define BufferSize 10

//user thread to select a random size
int basket[6] = {1000,2000,4000,8000,16000,32000};
//memory pointers
void *mPtr;

//Mutex & semaphore
pthread_mutex_t mutex_req;
pthread_mutex_t mutex_free;
sem_t threadSem;
sem_t ReqSem;
sem_t FreeSem;

//Defintion of Memory Block
typedef struct mBlockStruct{
  int size; //size of the memory block
  bool occupied; //buffer to place items produced
  long threadIdOccupied;
  struct mBlockStruct *nextBlock; //pointer to next block
}mBlock;

//first memory block intialization
mBlock *firstBlock = NULL;

//size allocated over all
int sizeTracker = 0;

void pushOnMBlock(int sizeTmp)
{
   mBlock *current = firstBlock;
   while(current->nextBlock!=NULL)
   {
     current = current->nextBlock;
   }
   current->nextBlock = (mBlock *)malloc(sizeof(mBlock));
   current->nextBlock->size=sizeTmp;
   current->nextBlock->nextBlock=NULL;
}

void printMBlock()
{
   mBlock *current = firstBlock;
   while(current->nextBlock!=NULL)
   {
     printf("Block Size:%d, Occupied:%s,By User:%d\n",current->size,current->occupied,current-
>threadIdOccupied);
     current = current->nextBlock;
   }
}

//Defintion of Request Queue
typedef struct {
  int in;  //Number of Requests on queue
  int out; //Number of Requests taken
  long threadIdRequested[BufferSize]; //buffer to place threadID
  int sizeRequested[BufferSize]; //buffer to place size
}ReqQue;

//Intialize Request queue
ReqQue ReqQueBuf = {0,0,{0},{0}};

//Defintion of Free Queue
typedef struct {
  int in;  //Number of Requests on queue
  int out; //Number of Requests taken
  long threadIdRequested[BufferSize]; //buffer to place threadID
}FreeQue;

//Intialize Free queue
FreeQue FreeQueBuf = {0,0,{0}};

//To allocate the memory block
void Best_Fit(int sizeReq, long threadIdReq)
{
   if((sizeReq==0)||(threadIdReq==0))
   {
     return;
   }
   if(sizeReq>MaxBlockSize)
   {
     printf("Size Can't be Allcated.Maximum BlockSize is %d\n",MaxBlockSize);
     return;
   }
   mBlock *current = firstBlock;
   mBlock *smallCurrent = NULL;
   int minSizeBlock = INT_MAX;
   while(current->nextBlock!=NULL)
   {
     if(((current->size)>=sizeReq) && ((current->occupied) == false))
       {
         if((current->size)<=minSizeBlock)
            {
               smallCurrent = current;
               minSizeBlock = current->size;
            }
       }
     current = current->nextBlock;
   }
   if(smallCurrent != NULL)
     {
       smallCurrent->occupied = true;
       smallCurrent->threadIdOccupied = threadIdReq;
       printf("Requested Memory:%d. Allocated:%d for User:%d \n",sizeReq,smallCurrent->size,threadIdReq);
       return;
     }
   printf("Out of Memory\n");
}

//To free the memory block
void Free_mBlock(long threadIdReq)
{
  if(threadIdReq==0)
   {
     return;
   }
   mBlock *current = firstBlock;
   while(current->nextBlock!=NULL)
   {
     if((current->threadIdOccupied)==threadIdReq)
     {
       current->occupied = false;
       current->threadIdOccupied = 0;
       printf("Free Memory:%d for User:%d \n",current->size,threadIdReq);
       return;
     }
     current = current->nextBlock;
   }
   printf("Free Unsucessful\n");
}

//Memory requested by user
void memory_malloc(int sizeReq, long threadIdReq)
{
      //wait on request semephore
      sem_wait(&ReqSem);
      //wait for the critical area access
      pthread_mutex_lock(&mutex_req);
      //check if request queue is full
      if(!(((ReqQueBuf.in + 1) % BufferSize ) == ReqQueBuf.out))
        {
          //Place the request on buffer
          ReqQueBuf.threadIdRequested[ReqQueBuf.in] = threadIdReq;
          ReqQueBuf.sizeRequested[ReqQueBuf.in] = sizeReq;
          //printf("Size Requested:%d by User:%d \n", ReqQueBuf.sizeRequested
[ReqQueBuf.in],ReqQueBuf.threadIdRequested[ReqQueBuf.in]);
          //Increment the 'in' index
          ReqQueBuf.in = (ReqQueBuf.in + 1) % BufferSize;
        }
      else
        {
          printf("Queue is full when accessed by User:%d \n",threadIdReq);
        }
      //release the critical area access
      pthread_mutex_unlock(&mutex_req);
      sleep(1);
}

//Memory manager to process the memory allocation request
void Process_Request()
{
      int sizeReqt;
      long threadIdReqt;
      //wait for the critical area access
      pthread_mutex_lock(&mutex_req);
      //Check if request buffer is empty
      if(!(ReqQueBuf.in == ReqQueBuf.out))
        {
          printf("Processing the Request for User:%d \n",ReqQueBuf.threadIdRequested[ReqQueBuf.in]);
          //Consume the request
          sizeReqt = ReqQueBuf.sizeRequested[ReqQueBuf.in];
          threadIdReqt = ReqQueBuf.threadIdRequested[ReqQueBuf.in];
          //Increment the 'out' index
          ReqQueBuf.out = (ReqQueBuf.out + 1) % BufferSize;
        }
      //release the critical area access
      pthread_mutex_unlock(&mutex_req);
      //Post the request sem
      sem_post(&ReqSem);
      //Call the algorithm
      Best_Fit(sizeReqt,threadIdReqt);
      sleep(1);
}

//Memory free request by user
void memory_free(long threadIdFree)
{
        //wait on free semephore
      sem_wait(&FreeSem);
      //wait for the critical area access
      pthread_mutex_lock(&mutex_free);
      //check if request queue is full
      if(!(((FreeQueBuf.in + 1) % BufferSize ) == FreeQueBuf.out))
        {
          //Place the request on buffer
          FreeQueBuf.threadIdRequested[FreeQueBuf.in] = threadIdFree;
          //printf("Size Free Requested by User:%d \n", threadIdFree);
          //Increment the 'in' index
          FreeQueBuf.in = (FreeQueBuf.in + 1) % BufferSize;
        }
      else
        {
          printf("Queue is full when accessed by User:%d \n", threadIdFree);
        }
      //release the critical area access
      pthread_mutex_unlock(&mutex_free);
      sleep(1);
}

//Memory Manager to process free request
void Process_Free()
{
      long threadIdReqt;
      //wait for the critical area access
      pthread_mutex_lock(&mutex_free);
      //Check if request buffer is empty
      if(!(FreeQueBuf.in == FreeQueBuf.out))
        {
          //printf("Processing the Free Request for User:%d \n",FreeQueBuf.threadIdRequested
[FreeQueBuf.in]);
          //Consume the request
          threadIdReqt = FreeQueBuf.threadIdRequested[FreeQueBuf.in];
          //Increment the 'out' index
          FreeQueBuf.out = (FreeQueBuf.out + 1) % BufferSize;
        }
      //release the critical area access
      pthread_mutex_unlock(&mutex_free);
      //Post the request sem
      sem_post(&FreeSem);
      Free_mBlock(threadIdReqt);
      sleep(1);
}

//Memory Manager
void *Manage(void* id)
{

        sleep(1);
        //printf("MMU:%d \n",(long)id);
        while(1)
        {
        Process_Request();
        Process_Free();
        }
}

//Request from user to allocate memory
void *Request(void* id)
{
        //printf("User:%d \n",(long)id);
        while(1)
        {
         int randNo = rand()%6;
        memory_malloc(basket[randNo],(long)id);
        sleep(10);
        memory_free((long)id);
        }
}


int main(int argc, char *argv[])
{
  //To get the number of users from cmd line
  unsigned int userNo = atoi(argv[1]);

  //intialize memory block
  firstBlock = (mBlock *)malloc(sizeof(mBlock));
  firstBlock->size = 1000;
  firstBlock->occupied = false;
  firstBlock->threadIdOccupied =0;
  firstBlock->nextBlock = NULL;
  sizeTracker = sizeTracker + 1000;
  //Allocate memory for the MMS
  mPtr = malloc(memorySize);
  //printf("mPtr:%d\n",mPtr);
  while(!(sizeTracker == memorySize))
  {
    int randNo = rand()%5;
    if(basket[randNo]<=(memorySize-sizeTracker))
    {
       pushOnMBlock(basket[randNo]);
       sizeTracker = sizeTracker+ basket[randNo];
    }
  }
  printf("sizeTracker:%d\n",sizeTracker);
  printMBlock();
  //Define MMU number
  unsigned int MMUNo = 1;
  pthread_t p[MMUNo];
  pthread_t* b;

  //Intialize mutex and two semaphores
  int e1 = pthread_mutex_init(&mutex_req, NULL);
  int e2 = pthread_mutex_init(&mutex_free, NULL);
  int e3 = sem_init(&ReqSem, 0, BufferSize-2);
  int e4 = sem_init(&FreeSem, 0, BufferSize-2);

  //Notify if failed
  if(e1!=0||e2!=0||e3!=0||e4!=0)
    printf("Intialization Error");

  b = malloc(userNo*sizeof(pthread_t));

  long i;
  //Create MMU threads
  for(i=0;i<MMUNo;i++)
    pthread_create(&p[i], NULL, Manage, (void*) i+1);

  //Create user threads
  for(i=0;i<userNo;i++)
    pthread_create(&b[i], NULL, Request, (void*) i+1);

  //Wait for MMU and user threads to finish
  for(i=0;i<MMUNo;i++)
    pthread_join(p[i], NULL);
  for(i=0;i<userNo;i++)
    pthread_join(b[i], NULL);

  //Destroy mutex and semaphores
  pthread_mutex_destroy(&mutex_req);
  pthread_mutex_destroy(&mutex_free);
  sem_destroy(&ReqSem);
  sem_destroy(&FreeSem);
  return 0;
}
```

```c
#include<stdio.h>
#include<stdbool.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
#include<semaphore.h>
#include<signal.h>
#include<limits.h>

//memory for MMS to manage in Bytes
#define memorySize 128000
#define MaxBlockSize 16000
//Define the size of the queue
#define BufferSize 10

//user thread to select a random size
int basket[6] = {1000,2000,4000,8000,16000,32000};
//memory pointers
void *mPtr;

//Mutex & semaphore
pthread_mutex_t mutex_req;
pthread_mutex_t mutex_free;
sem_t threadSem;
sem_t ReqSem;
sem_t FreeSem;

//Defintion of Memory Block
typedef struct mBlockStruct{
  int size; //size of the memory block
  bool occupied; //buffer to place items produced
  long threadIdOccupied;
  struct mBlockStruct *nextBlock; //pointer to next block
}mBlock;

//first memory block intialization
mBlock *firstBlock = NULL;

//size allocated over all
int sizeTracker = 0;

void pushOnMBlock(int sizeTmp)
{
   mBlock *current = firstBlock;
   while(current->nextBlock!=NULL)
   {
     current = current->nextBlock;
   }
   current->nextBlock = (mBlock *)malloc(sizeof(mBlock));
   current->nextBlock->size=sizeTmp;
   current->nextBlock->nextBlock=NULL;
}

void printMBlock()
{
   mBlock *current = firstBlock;
   while(current->nextBlock!=NULL)
   {
     printf("Block Size:%d, Occupied:%s,By User:%d\n",current->size,current->occupied,current-
>threadIdOccupied);
     current = current->nextBlock;
   }
}

//Defintion of Request Queue
typedef struct {
  int in;  //Number of Requests on queue
  int out; //Number of Requests taken
  long threadIdRequested[BufferSize]; //buffer to place threadID
  int sizeRequested[BufferSize]; //buffer to place size
}ReqQue;

//Intialize Request queue
ReqQue ReqQueBuf = {0,0,{0},{0}};

//Defintion of Free Queue
typedef struct {
  int in;  //Number of Requests on queue
  int out; //Number of Requests taken
  long threadIdRequested[BufferSize]; //buffer to place threadID
}FreeQue;

//Intialize Free queue
FreeQue FreeQueBuf = {0,0,{0}};

//To allocate the memory block
void Worst_Fit(int sizeReq, long threadIdReq)
{
   if((sizeReq==0)||(threadIdReq==0))
   {
     return;
   }
   if(sizeReq>MaxBlockSize)
   {
     printf("Size Can't be Allcated.Maximum BlockSize is %d\n",MaxBlockSize);
     return;
   }
   mBlock *current = firstBlock;
   mBlock *bigCurrent = NULL;
   int maxSizeBlock = INT_MIN;
   while(current->nextBlock!=NULL)
   {
     if(((current->size)>=sizeReq) && ((current->occupied) == false))
       {
         if((current->size)>=maxSizeBlock)
            {
               bigCurrent = current;
               maxSizeBlock = current->size;
            }
       }
     current = current->nextBlock;
   }
   if(bigCurrent != NULL)
     {
       bigCurrent->occupied = true;
       bigCurrent->threadIdOccupied = threadIdReq;
       printf("Requested Memory:%d. Allocated:%d for User:%d \n",sizeReq,bigCurrent->size,threadIdReq);
       return;
     }
   printf("Out of Memory\n");
}

//To free the memory block
void Free_mBlock(long threadIdReq)
{
  if(threadIdReq==0)
   {
     return;
   }
   mBlock *current = firstBlock;
   while(current->nextBlock!=NULL)
   {
     if((current->threadIdOccupied)==threadIdReq)
     {
```

```c
       current->occupied = false;
       current->threadIdOccupied = 0;
       printf("Free Memory:%d for User:%d \n",current->size,threadIdReq);
       return;
     }
     current = current->nextBlock;
   }
   printf("Free Unsucessful\n");
}

//Memory requested by user
void memory_malloc(int sizeReq, long threadIdReq)
{
      //wait on request semephore
      sem_wait(&ReqSem);
      //wait for the critical area access
      pthread_mutex_lock(&mutex_req);
      //check if request queue is full
      if(!(((ReqQueBuf.in + 1) % BufferSize ) == ReqQueBuf.out))
        {
          //Place the request on buffer
          ReqQueBuf.threadIdRequested[ReqQueBuf.in] = threadIdReq;
          ReqQueBuf.sizeRequested[ReqQueBuf.in] = sizeReq;
          //printf("Size Requested:%d by User:%d \n", ReqQueBuf.sizeRequested
[ReqQueBuf.in],ReqQueBuf.threadIdRequested[ReqQueBuf.in]);
          //Increment the 'in' index
          ReqQueBuf.in = (ReqQueBuf.in + 1) % BufferSize;
        }
      else
        {
          printf("Queue is full when accessed by User:%d \n",threadIdReq);
        }
      //release the critical area access
      pthread_mutex_unlock(&mutex_req);
      sleep(1);
}

//Memory manager to process the memory allocation request
void Process_Request()
{
      int sizeReqt;
      long threadIdReqt;
      //wait for the critical area access
      pthread_mutex_lock(&mutex_req);
      //Check if request buffer is empty
      if(!(ReqQueBuf.in == ReqQueBuf.out))
        {
          printf("Processing the Request for User:%d \n",ReqQueBuf.threadIdRequested[ReqQueBuf.in]);
          //Consume the request
          sizeReqt = ReqQueBuf.sizeRequested[ReqQueBuf.in];
          threadIdReqt = ReqQueBuf.threadIdRequested[ReqQueBuf.in];
          //Increment the 'out' index
          ReqQueBuf.out = (ReqQueBuf.out + 1) % BufferSize;
        }
      //release the critical area access
      pthread_mutex_unlock(&mutex_req);
      //Post the request sem
      sem_post(&ReqSem);
      //Call the algorithm
      Worst_Fit(sizeReqt,threadIdReqt);
      sleep(1);
}

//Memory free request by user
void memory_free(long threadIdFree)
{
        //wait on free semephore
      sem_wait(&FreeSem);
      //wait for the critical area access
      pthread_mutex_lock(&mutex_free);
      //check if request queue is full
      if(!(((FreeQueBuf.in + 1) % BufferSize ) == FreeQueBuf.out))
        {
          //Place the request on buffer
          FreeQueBuf.threadIdRequested[FreeQueBuf.in] = threadIdFree;
          //printf("Size Free Requested by User:%d \n", threadIdFree);
          //Increment the 'in' index
          FreeQueBuf.in = (FreeQueBuf.in + 1) % BufferSize;
        }
      else
        {
          printf("Queue is full when accessed by User:%d \n", threadIdFree);
        }
      //release the critical area access
      pthread_mutex_unlock(&mutex_free);
      sleep(1);
}

//Memory Manager to process free request
void Process_Free()
{
      long threadIdReqt;
      //wait for the critical area access
      pthread_mutex_lock(&mutex_free);
      //Check if request buffer is empty
      if(!(FreeQueBuf.in == FreeQueBuf.out))
        {
          //printf("Processing the Free Request for User:%d \n",FreeQueBuf.threadIdRequested
[FreeQueBuf.in]);
          //Consume the request
          threadIdReqt = FreeQueBuf.threadIdRequested[FreeQueBuf.in];
          //Increment the 'out' index
          FreeQueBuf.out = (FreeQueBuf.out + 1) % BufferSize;
        }
      //release the critical area access
      pthread_mutex_unlock(&mutex_free);
      //Post the request sem
      sem_post(&FreeSem);
      Free_mBlock(threadIdReqt);
      sleep(1);
}

//Memory Manager
void *Manage(void* id)
{

        sleep(1);
        //printf("MMU:%d \n",(long)id);
        while(1)
        {
        Process_Request();
        Process_Free();
        }
}

//Request from user to allocate memory
void *Request(void* id)
{
        //printf("User:%d \n",(long)id);
        while(1)
        {
         int randNo = rand()%6;
        memory_malloc(basket[randNo],(long)id);
        sleep(10);
        memory_free((long)id);
        }
}


int main(int argc, char *argv[])
{
  //To get the number of users from cmd line
  unsigned int userNo = atoi(argv[1]);

  //intialize memory block
  firstBlock = (mBlock *)malloc(sizeof(mBlock));
  firstBlock->size = 1000;
  firstBlock->occupied = false;
  firstBlock->threadIdOccupied =0;
  firstBlock->nextBlock = NULL;
  sizeTracker = sizeTracker + 1000;
  //Allocate memory for the MMS
  mPtr = malloc(memorySize);
  //printf("mPtr:%d\n",mPtr);
  while(!(sizeTracker == memorySize))
  {
    int randNo = rand()%5;
    if(basket[randNo]<=(memorySize-sizeTracker))
    {
       pushOnMBlock(basket[randNo]);
       sizeTracker = sizeTracker+ basket[randNo];
    }
  }
  printf("sizeTracker:%d\n",sizeTracker);
  printMBlock();
  //Define MMU number
  unsigned int MMUNo = 1;
  pthread_t p[MMUNo];
  pthread_t* b;

  //Intialize mutex and two semaphores
  int e1 = pthread_mutex_init(&mutex_req, NULL);
  int e2 = pthread_mutex_init(&mutex_free, NULL);
  int e3 = sem_init(&ReqSem, 0, BufferSize-2);
  int e4 = sem_init(&FreeSem, 0, BufferSize-2);

  //Notify if failed
  if(e1!=0||e2!=0||e3!=0||e4!=0)
    printf("Intialization Error");

  b = malloc(userNo*sizeof(pthread_t));

  long i;
  //Create MMU threads
  for(i=0;i<MMUNo;i++)
    pthread_create(&p[i], NULL, Manage, (void*) i+1);

  //Create user threads
  for(i=0;i<userNo;i++)
    pthread_create(&b[i], NULL, Request, (void*) i+1);

  //Wait for MMU and user threads to finish
  for(i=0;i<MMUNo;i++)
    pthread_join(p[i], NULL);
  for(i=0;i<userNo;i++)
    pthread_join(b[i], NULL);

  //Destroy mutex and semaphores
  pthread_mutex_destroy(&mutex_req);
  pthread_mutex_destroy(&mutex_free);
  sem_destroy(&ReqSem);
  sem_destroy(&FreeSem);
  return 0;
}
```

```
#include<stdio.h>
#include<stdbool.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
#include<semaphore.h>
#include<signal.h>

//memory for MMS to manage in Bytes
#define memorySize 128000
#define MaxBlockSize 16000
//Define the size of the queue
#define BufferSize 10

//user thread to select a random size
int basket[7] = {1000,2000,4000,8000,16000,32000,32000};
//memory pointers
void *mPtr;

//Mutex & semaphore
pthread_mutex_t mutex_req;
pthread_mutex_t mutex_free;
sem_t threadSem;
sem_t ReqSem;
sem_t FreeSem;

//Defintion of Memory Block
typedef struct mBlockStruct{
  int size; //size of the memory block
  bool occupied; //buffer to place items produced
  long threadIdOccupied;
  struct mBlockStruct *nextBlock; //pointer to next block
}mBlock;

//first memory block intialization
mBlock *firstBlock = NULL;

//size allocated over all
int sizeTracker = 0;

void pushOnMBlock(int sizeTmp)
{
   mBlock *current = firstBlock;
   while(current->nextBlock!=NULL)
   {
     current = current->nextBlock;
   }
   current->nextBlock = (mBlock *)malloc(sizeof(mBlock));
   current->nextBlock->size=sizeTmp;
   current->nextBlock->nextBlock=NULL;
}

void printMBlock()
{
   mBlock *current = firstBlock;
   while(current->nextBlock!=NULL)
   {
     printf("Block Size:%d, Occupied:%s,By User:%d\n",current->size,current->occupied,current-
>threadIdOccupied);
     current = current->nextBlock;
   }
}

//Defintion of Request Queue
typedef struct {
  int in;  //Number of Requests on queue
  int out; //Number of Requests taken
  long threadIdRequested[BufferSize]; //buffer to place threadID
  int sizeRequested[BufferSize]; //buffer to place size
}ReqQue;

//Intialize Request queue
ReqQue ReqQueBuf = {0,0,{0},{0}};

//Defintion of Free Queue
typedef struct {
  int in;  //Number of Requests on queue
  int out; //Number of Requests taken
  long threadIdRequested[BufferSize]; //buffer to place threadID
}FreeQue;

//Intialize Free queue
FreeQue FreeQueBuf = {0,0,{0}};

//To allocate the memory block
void First_Fit(int sizeReq, long threadIdReq)
{
   if((sizeReq==0)||(threadIdReq==0))
   {
     return;
   }
   /*if(sizeReq>MaxBlockSize)
   {
     printf("Size Can't be Allcated.Maximum BlockSize is %d\n",MaxBlockSize);
     return;
   }*/
   mBlock *current = firstBlock;
   mBlock *currentTmp[memorySize/MaxBlockSize]={NULL};
   int i =0;
   while(current->nextBlock!=NULL)
   {
        //collecting free blocks
     if((current->occupied) == false)
        {
            currentTmp[i++] = current;
        }
        //Finding the right block
     if(((current->size)>=sizeReq) && ((current->occupied) == false))
     {
       current->occupied = true;
       current->threadIdOccupied = threadIdReq;
       printf("Allocated:%d for User:%d \n",current->size,threadIdReq);
       return;
     }
     current = current->nextBlock;
   }
   //Defrag Block iterate through empty block to assign
   int j=0;
   int tmpSize = 0;
   for(j=0;j<(memorySize/MaxBlockSize);j++)
   {
        if(currentTmp[j]!=NULL)
        {
        currentTmp[j]->occupied = true;
        currentTmp[j]->threadIdOccupied = threadIdReq;
        printf("Defrag-Allocated:%d for User:%d OutOf:%d \n",currentTmp[j]->size,threadIdReq,sizeReq);
        tmpSize = tmpSize + currentTmp[j]->size;
        }
        if(tmpSize>=sizeReq)
        {
            return;
        }
   }
   //If Defrag not met kick out lower priorty thread
   //iterate list again
   current = firstBlock;
   while(current->nextBlock!=NULL)
   {
        //higher rank user has higher priorty
        if((current->threadIdOccupied)<threadIdReq)
        {
           printf("Kicked Out User:%d to Allocate:%d for User:%d Outof:%d \n",current-
>threadIdOccupied,current->size,threadIdReq,sizeReq);
           current->threadIdOccupied = threadIdReq;
           tmpSize = tmpSize + current->size;
        }
        if(tmpSize>=sizeReq)
        {
            return;
        }
        current = current->nextBlock;
   }
   //Out of memory
   printf("Out of Memory\n");
}

//To free the memory block
void Free_mBlock(long threadIdReq)
{
  bool flag = false;
  if(threadIdReq==0)
   {
     return;
   }
   mBlock *current = firstBlock;
   while(current->nextBlock!=NULL)
   {
     if((current->threadIdOccupied)==threadIdReq)
     {
       current->occupied = false;
       current->threadIdOccupied = 0;
       printf("Free Memory:%d for User:%d \n",current->size,threadIdReq);
       flag = true;
     }
     current = current->nextBlock;
   }
   if(flag)
   {
     return;
   }
   printf("Free Unsucessful\n");
}

//Memory requested by user
void memory_malloc(int sizeReq, long threadIdReq)
{
      //wait on request semephore
      sem_wait(&ReqSem);
      //wait for the critical area access
      pthread_mutex_lock(&mutex_req);
      //check if request queue is full
      if(!(((ReqQueBuf.in + 1) % BufferSize ) == ReqQueBuf.out))
        {
          //Place the request on buffer
          ReqQueBuf.threadIdRequested[ReqQueBuf.in] = threadIdReq;
          ReqQueBuf.sizeRequested[ReqQueBuf.in] = sizeReq;
          //printf("Size Requested:%d by User:%d \n", ReqQueBuf.sizeRequested
[ReqQueBuf.in],ReqQueBuf.threadIdRequested[ReqQueBuf.in]);
          //Increment the 'in' index
          ReqQueBuf.in = (ReqQueBuf.in + 1) % BufferSize;
        }
      else
        {
          printf("Queue is full when accessed by User:%d \n",threadIdReq);
        }
      //release the critical area access
      pthread_mutex_unlock(&mutex_req);
      sleep(1);
}

//Memory manager to process the memory allocation request
void Process_Request()
{
      int sizeReqt;
      long threadIdReqt;
      //wait for the critical area access
      pthread_mutex_lock(&mutex_req);
      //Check if request buffer is empty
      if(!(ReqQueBuf.in == ReqQueBuf.out))
        {
          printf("Processing the Request for User:%d \n",ReqQueBuf.threadIdRequested[ReqQueBuf.in]);
          //Consume the request
          sizeReqt = ReqQueBuf.sizeRequested[ReqQueBuf.in];
          threadIdReqt = ReqQueBuf.threadIdRequested[ReqQueBuf.in];
          //Increment the 'out' index
          ReqQueBuf.out = (ReqQueBuf.out + 1) % BufferSize;
        }
      //release the critical area access
      pthread_mutex_unlock(&mutex_req);
      //Post the request sem
      sem_post(&ReqSem);
      //Call the algorithm
      First_Fit(sizeReqt,threadIdReqt);
      sleep(1);
}

//Memory free request by user
void memory_free(long threadIdFree)
{
        //wait on free semephore
      sem_wait(&FreeSem);
      //wait for the critical area access
      pthread_mutex_lock(&mutex_free);
      //check if request queue is full
      if(!(((FreeQueBuf.in + 1) % BufferSize ) == FreeQueBuf.out))
        {
          //Place the request on buffer
          FreeQueBuf.threadIdRequested[FreeQueBuf.in] = threadIdFree;
          //printf("Size Free Requested by User:%d \n", threadIdFree);
          //Increment the 'in' index
          FreeQueBuf.in = (FreeQueBuf.in + 1) % BufferSize;
        }
      else
        {
          printf("Queue is full when accessed by User:%d \n", threadIdFree);
        }
      //release the critical area access
      pthread_mutex_unlock(&mutex_free);
      sleep(1);
}

//Memory Manager to process free request
void Process_Free()
{
      long threadIdReqt;
      //wait for the critical area access
      pthread_mutex_lock(&mutex_free);
      //Check if request buffer is empty
      if(!(FreeQueBuf.in == FreeQueBuf.out))
        {
          //printf("Processing the Free Request for User:%d \n",FreeQueBuf.threadIdRequested
[FreeQueBuf.in]);
          //Consume the request
          threadIdReqt = FreeQueBuf.threadIdRequested[FreeQueBuf.in];
          //Increment the 'out' index
          FreeQueBuf.out = (FreeQueBuf.out + 1) % BufferSize;
        }
      //release the critical area access
      pthread_mutex_unlock(&mutex_free);
      //Post the request sem
      sem_post(&FreeSem);
      Free_mBlock(threadIdReqt);
      sleep(1);
}

//Memory Manager
void *Manage(void* id)
{

        sleep(1);
        //printf("MMU:%d \n",(long)id);
        while(1)
        {
        Process_Request();
        Process_Free();
        }
}

//Request from user to allocate memory
void *Request(void* id)
{
        //printf("User:%d \n",(long)id);
        while(1)
        {
         int randNo = rand()%7;
        memory_malloc(basket[randNo],(long)id);
        sleep(4);
        memory_free((long)id);
        }
}


int main(int argc, char *argv[])
{
  //To get the number of users from cmd line
  unsigned int userNo = atoi(argv[1]);

  //intialize memory block
  firstBlock = (mBlock *)malloc(sizeof(mBlock));
  firstBlock->size = 1000;
  firstBlock->occupied = false;
  firstBlock->threadIdOccupied =0;
  firstBlock->nextBlock = NULL;
  sizeTracker = sizeTracker + 1000;
  //Allocate memory for the MMS
  mPtr = malloc(memorySize);
  //printf("mPtr:%d\n",mPtr);
  while(!(sizeTracker == memorySize))
  {
    int randNo = rand()%5;
    if(basket[randNo]<=(memorySize-sizeTracker))
    {
       pushOnMBlock(basket[randNo]);
       sizeTracker = sizeTracker+ basket[randNo];
    }
  }
  printf("sizeTracker:%d\n",sizeTracker);
  printMBlock();
  //Define MMU number
  unsigned int MMUNo = 1;
  pthread_t p[MMUNo];
  pthread_t* b;

  //Intialize mutex and two semaphores
  int e1 = pthread_mutex_init(&mutex_req, NULL);
  int e2 = pthread_mutex_init(&mutex_free, NULL);
  int e3 = sem_init(&ReqSem, 0, BufferSize-2);
  int e4 = sem_init(&FreeSem, 0, BufferSize-2);

  //Notify if failed
  if(e1!=0||e2!=0||e3!=0||e4!=0)
    printf("Intialization Error");

  b = malloc(userNo*sizeof(pthread_t));

  long i;
  //Create MMU threads
  for(i=0;i<MMUNo;i++)
    pthread_create(&p[i], NULL, Manage, (void*) i+1);

  //Create user threads
  for(i=0;i<userNo;i++)
    pthread_create(&b[i], NULL, Request, (void*) i+1);

  //Wait for MMU and user threads to finish
  for(i=0;i<MMUNo;i++)
    pthread_join(p[i], NULL);
  for(i=0;i<userNo;i++)
    pthread_join(b[i], NULL);

  //Destroy mutex and semaphores
  pthread_mutex_destroy(&mutex_req);
  pthread_mutex_destroy(&mutex_free);
  sem_destroy(&ReqSem);
  sem_destroy(&FreeSem);
  return 0;
}
```