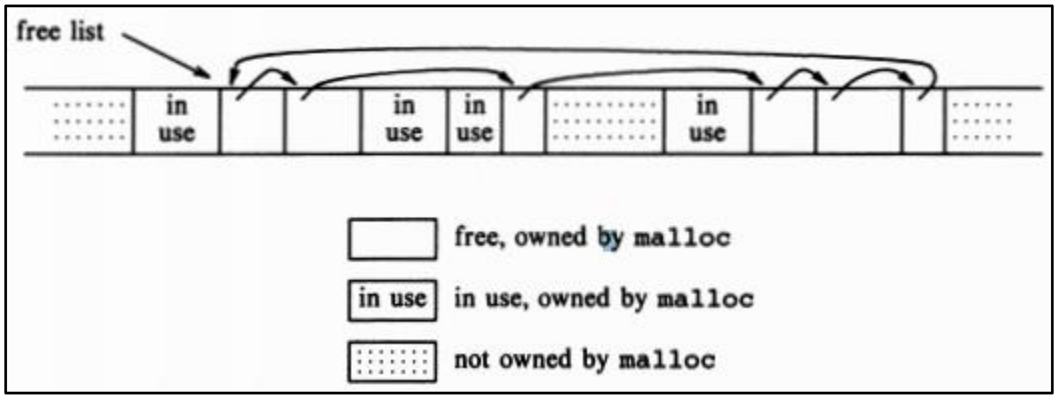
MALLOC Call Tracing:

Types of Memory:

- Stack memory - Allocations and deallocations are done implicitly by the compiler. (Static memory allocation)
- Heap memory - Allocations and deallocations are done explicitly by the programmer. (Dynamic memory allocation) (malloc).

Malloc structure details:

- malloc will request space from the operating system as needed.
- Since other activities in the program may also request space without calling this allocator, the space that malloc manages may not be contiguous.
- Thus, its free storage is kept as a list of free blocks.
- Each block contains a size, a pointer to the next block, and the space.
- The blocks are kept in order of increasing storage address, and the last block (highest address) points to the first.



- When a request is made, the free list is scanned until a big-enough block is found and using different algorithms it is given a block of memory to fulfill its request.

Input and output:

- Malloc takes size as an input return a pointer of the newly created memory.
- Malloc returns “NULL” or 0 if it fails to allocate memory.

Declaration:

```
int *x = (int *) malloc(sizeof(int));
```

<u>Tracing:</u>

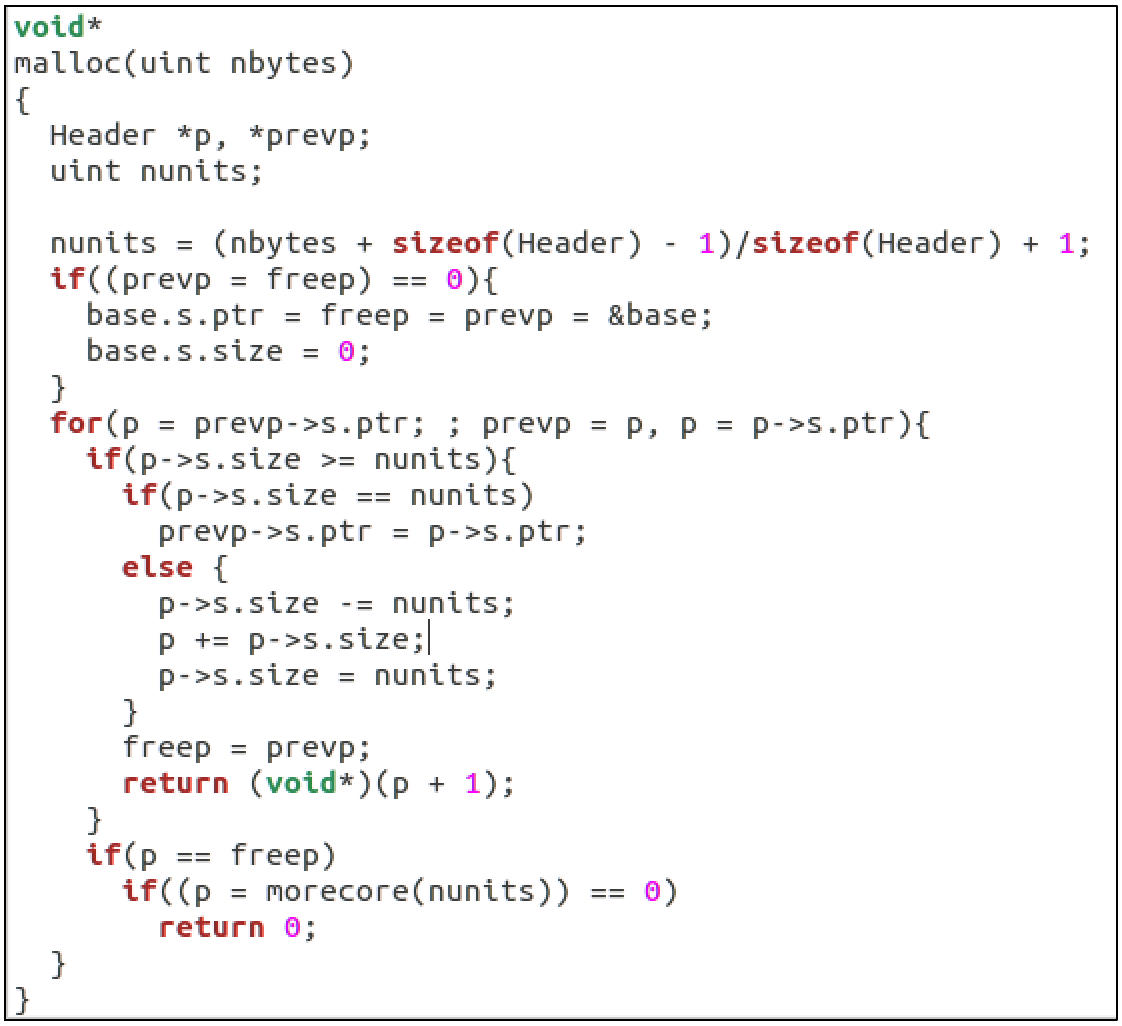
- Malloc belongs to the stdlib.h library file.
- Execution of the malloc starts from the user program. Eg: (sh.c)

```
cmd = malloc(sizeof(*cmd));
```

- Malloc is defined in the "**user.h**" file.

```
void* malloc(uint);
```

- Implementation of malloc function is present in the "umalloc.c" file.



```
void*
malloc(uint nbytes)
{
  Header *p, *prevp;
  uint nunits;

  nunits = (nbytes + sizeof(Header) - 1)/sizeof(Header) + 1;
  if((prevp = freep) == 0){
    base.s.ptr = freep = prevp = &base;
    base.s.size = 0;
  }
  for(p = prevp->s.ptr; ; prevp = p, p = p->s.ptr){
    if(p->s.size >= nunits){
      if(p->s.size == nunits)
        prevp->s.ptr = p->s.ptr;
      else {
        p->s.size -= nunits;
        p += p->s.size;
        p->s.size = nunits;
      }
      freep = prevp;
      return (void*)(p + 1);
    }
    if(p == freep)
      if((p = morecore(nunits)) == 0)
        return 0;
  }
}
```

- Malloc creates two pointers (p, prevp) as the type header.

```
typedef long Align;   /* for alignment to long boundary */

union header {        /* block header: */
    struct {
        union header *ptr; /* next block if on free list */
        unsigned size;     /* size of this block */
    } s;
    Align x;          /* force alignment of blocks */
};

typedef union header Header;
```

Explanation of implementation of malloc () call:

- The header type has a pointer(ptr) which points to next free bloc in the list.
- A size attribute for specifying the size of the current block.
- Align x for alignment of the blocks.
- When a malloc is called with a size of n bytes, it doesn't get a block size of exactly n bytes rather it has the combined size of header and size of n bytes which is stored in nunits attribute.

```
nunits = (nbytes + sizeof(Header) - 1)/sizeof(Header) + 1;
```

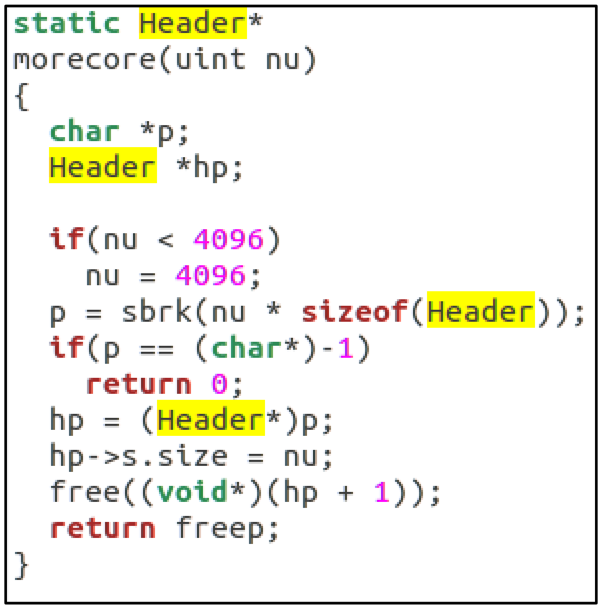
- The variable base is used to get started.
- If freep is NULL, as it is at the first call of malloc, then a degenerate free list is created; it contains one block of size zero, and points to itself.

```
if((prevp = freep) == 0)
```

- When the free list is searched for a free block of adequate size begins at the point (freep) where the last block was found; this strategy helps keep the list homogeneous.
- If a too-big block is found, the tail end is returned to the user; in this way the header of the original needs only to have its size adjusted.
- In all cases, the pointer returned to the user points to the free space within the block, which begins one unit beyond the header.

```
if (p == freep)  /* wrapped around free list */
    if ((p = morecore(nunits)) == NULL)
        return NULL;   /* none left */
```

Morecore function:



```
static Header*
morecore(uint nu)
{
  char *p;
  Header *hp;

  if(nu < 4096)
    nu = 4096;
  p = sbrk(nu * sizeof(Header));
  if(p == (char*)-1)
    return 0;
  hp = (Header*)p;
  hp->s.size = nu;
  free((void*)(hp + 1));
  return freep;
}
```

- Morecore function takes size(nuints) as input and gives bigger block size if it exceeds 4Kbytes.
- The function morecore obtains storage from the operating system.
- Since asking the system for memory is a comparatively expensive operation, we don't want to do that on every call to malloc, so morecore requests at least 4096 units; this larger block will be chopped up as needed.
- After setting the size field, morecore inserts the additional memory into the arena by calling free.
- The UNIX system call sbrk (n) returns a pointer to n more bytes of storage. sbrk returns -1 if there was no space.
- The - 1 must be cast to char *.so it can be compared with the return value.
- Again, casts make the function relatively immune to the details of pointer representation on different machines. There is still one assumption, however, that pointers to different blocks returned by sbrk can be meaningfully compare.
- The return value of morecore function is freep.

Common errors in the usage of Malloc () call:

Forgetting to allocate memory:

- When we create a pointer an uninitialized destination pointer and we try to load the source content in destination we get a "**SEGMENT FAULT**" exception.

```
char *src = "hello";
char *dst;          // oops! unallocated
strcpy(dst, src);   // segfault and die
```

Not Allocating enough memory:

- When we don't allocate enough memory, it will override the contents present in the subsequence locations. (**BUFFER OVERFLOW**)

```
char *src = "hello";
char *dst = (char *) malloc(strlen(src)); // too small!
strcpy(dst, src); // work properly
```

Forgetting to initialize allocated memory:

- This occurs when you forgot to fill in some value into newly allotted data type. (**UNINITIALIZED READ**)

These operations are harmful and will result in a trap into the operating system.

Free ():

- Free () function is used to free up the allocated region.
- Free function takes a pointer as an argument that was returned from malloc and frees the memory.
- Important to note is that we are not specifying the number of bytes of memory rather just passing a pointer because the pointer has a header region and it has the size of memory to be freed, this is tracked by the memory allocation library.

Declaration:

```
int *x = malloc(10 * sizeof(int));
...
free(x);
```

Tracing Details:

- Free function belongs to stdlib.h library file.
- Calling of the free functions starts in the user space after obtaining the pointer from the malloc function.
- Definition of the free function is declared in the "**user.h**" file.

```
void free(void*);
```

- Implementation of the free function is present in the "**umalloc.c**" file.

```
void
free(void *ap)
{
  Header *bp, *p;

  bp = (Header*)ap - 1;
  for(p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
    if(p >= p->s.ptr && (bp > p || bp < p->s.ptr))
      break;
  if(bp + bp->s.size == p->s.ptr){
    bp->s.size += p->s.ptr->s.size;
    bp->s.ptr = p->s.ptr->s.ptr;
  } else
    bp->s.ptr = p->s.ptr;
  if(p + p->s.size == bp){
    p->s.size += bp->s.size;
    p->s.ptr = bp->s.ptr;
  } else
    p->s.ptr = bp;
  freep = p;
}
```

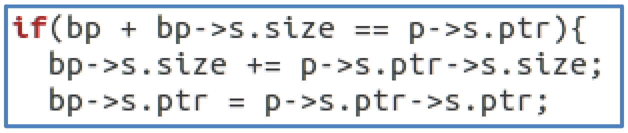
- It has two pointers p and bp of the type header.
- It scans the free list, starting at freep, looking for the place to insert the free block.
- This is either between two existing blocks or at one end of the list.
- In any case, if the block being freed is adjacent to either neighbor, the adjacent blocks are combined (important checking to do coalescing of free block. this technique reduces overhead).
- The only troubles are keeping the pointers pointing to the right things and the sizes correct.
- Points to the block header.

```
bp = (Header*)ap - 1;
```

- Frees block at start or end.

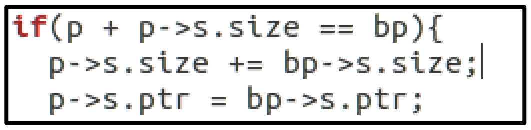
```
for(p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
  if(p >= p->s.ptr && (bp > p || bp < p->s.ptr))
    break;
```

- Joins to upper nbr.



```
if(bp + bp->s.size == p->s.ptr){
  bp->s.size += p->s.ptr->s.size;
  bp->s.ptr = p->s.ptr->s.ptr;
```

- Joins to the lower nbr.



```
if(p + p->s.size == bp){
  p->s.size += bp->s.size;
  p->s.ptr = bp->s.ptr;
```

<u>**Common errors in the usage of Free () call:**</u>

Forgetting to free memory: (Memory Leak)

- ➢ It occurs when you forgot to free memory. It is a slowly occurring process and finally results in out of memory at that point of time a restart is required.

Freeing memory before you are dome with it:(Dangling Pointer)

- ➢ It occurs when the memory is freed before a program is not finished using it.
- ➢ Subsequent use can crash the program or overwrite any valid address.

Freeing the memory twice:(Double free)

- ➢ It occurs when the program frees memory more than once. The result of doing this undefined.

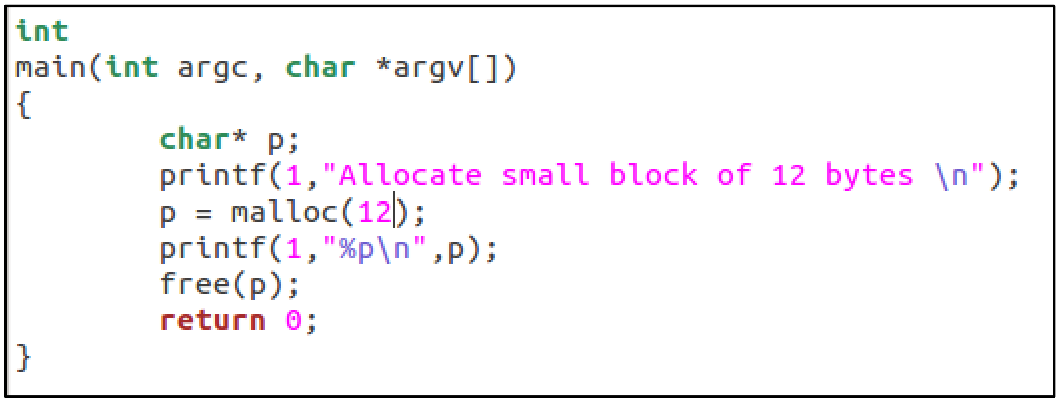
Calling free incorrectly:(Invalid frees)

- ➢ It occurs when you pass any other type of value other than a pointer to the free function. These are dangerous and should be avoided.

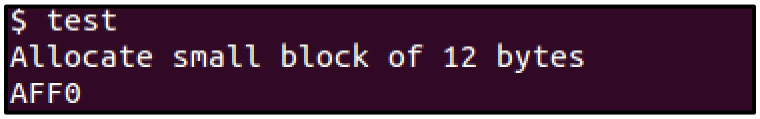
<u>Testing of Malloc () and Free ():</u>

Case 1:

- Allocating a block of size 12 bytes. This type of allocation is not preferred.

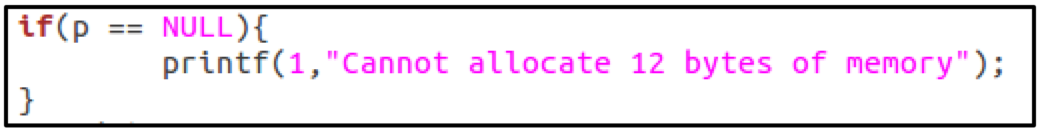


```
int
main(int argc, char *argv[])
{
        char* p;
        printf(1,"Allocate small block of 12 bytes \n");
        p = malloc(12);
        printf(1,"%p\n",p);
        free(p);
        return 0;
}
```



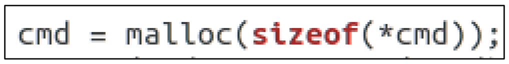
```
$ test
Allocate small block of 12 bytes
AFF0
```

- Exception handling on the above program can be specified as. If the malloc fails to allocate memory it will return null.



```
if(p == NULL){
        printf(1,"Cannot allocate 12 bytes of memory");
}
```

- Preferred declaration is

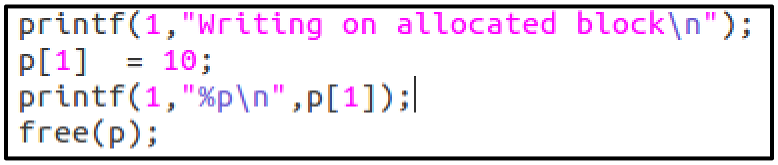


```
cmd = malloc(sizeof(*cmd));
```

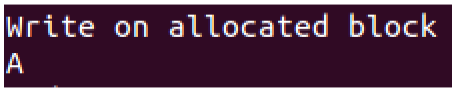
We use sizeof() operator to specify the size of the variable .This is done at compile time .

Case 2:

- Writing on the allocated block of memory.



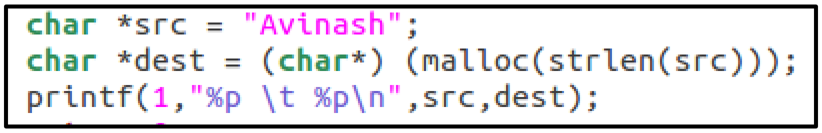
```
printf(1,"Writing on allocated block\n");
p[1]  = 10;
printf(1,"%p\n",p[1]);
free(p);
```



```
Write on allocated block
A
```

Case 3:

- Not allocating enough space.

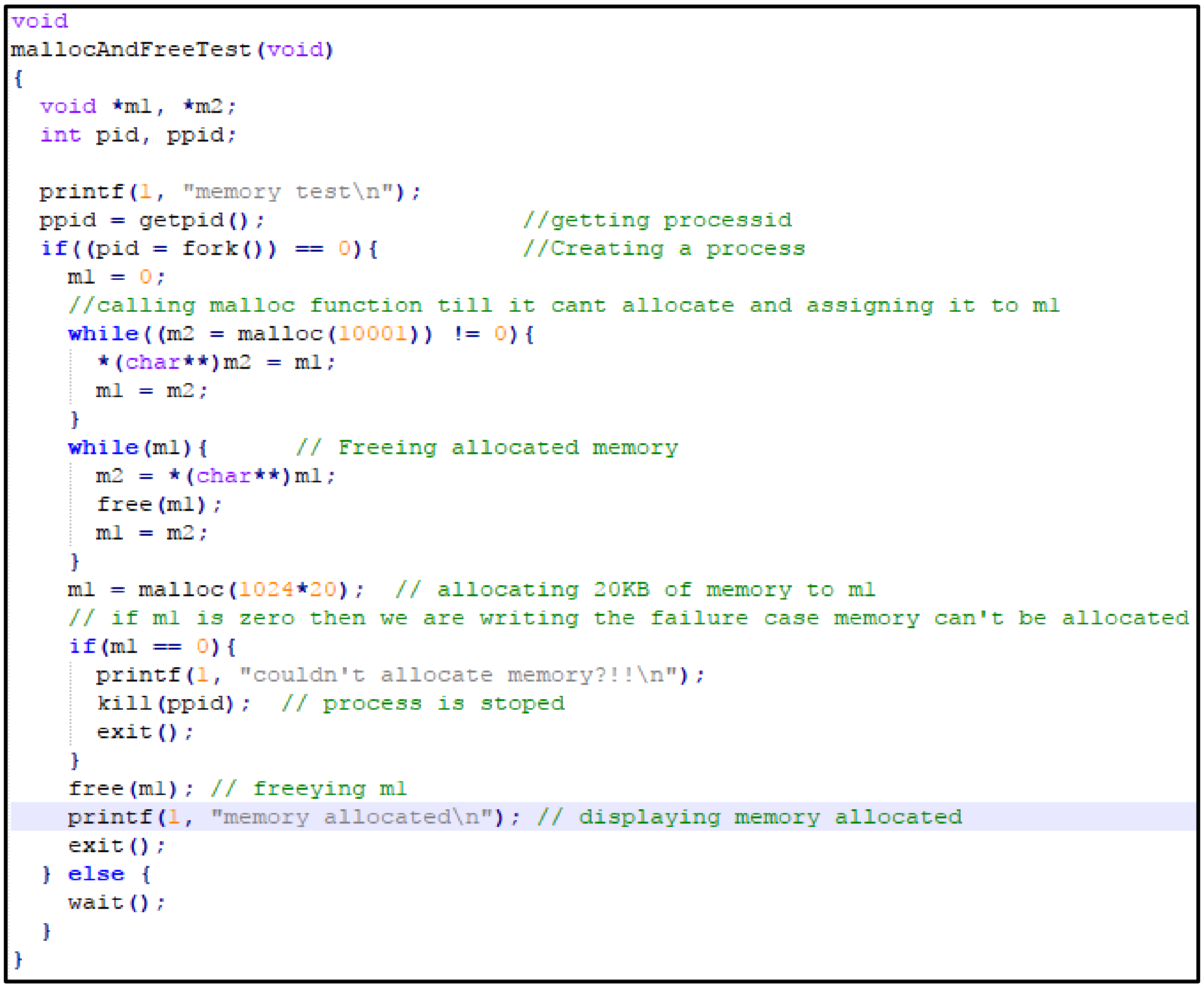


```
char *src = "Avinash";
char *dest = (char*) (malloc(strlen(src)));
printf(1,"%p \t %p\n",src,dest);
```

- Results in segmentation fault exception correct declaration is “**char *dest = (char*) (malloc(strlen(src)) +1)**; “

Case 4:

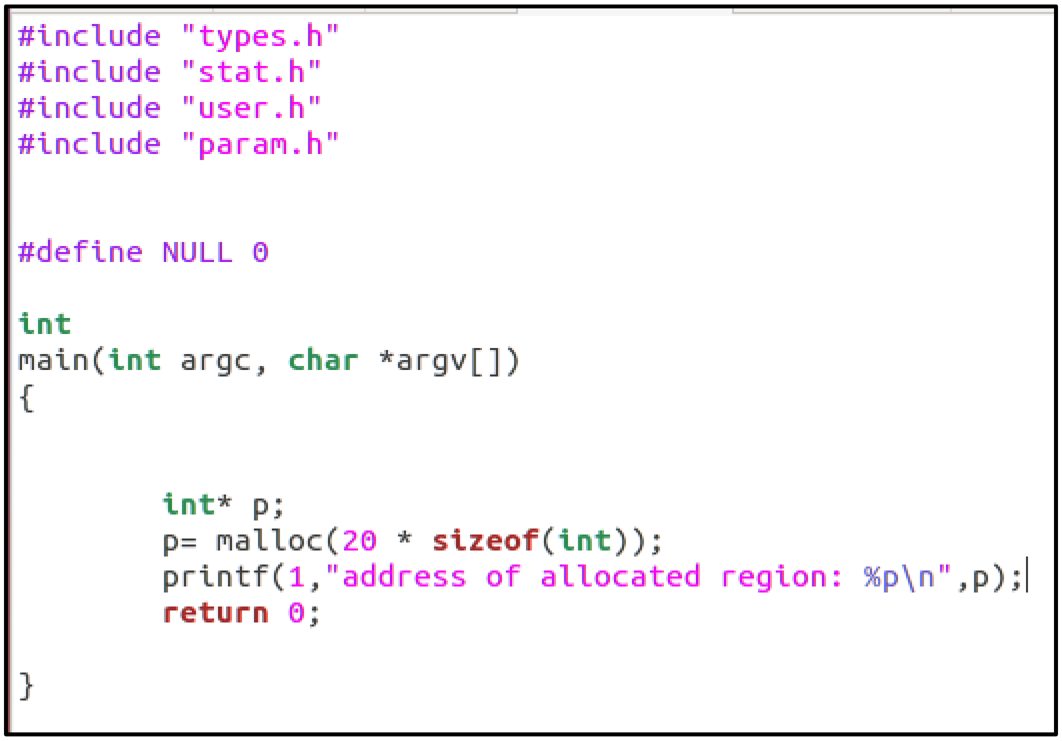
- Creating a process by calling fork() method, allocating and deallocating memory.



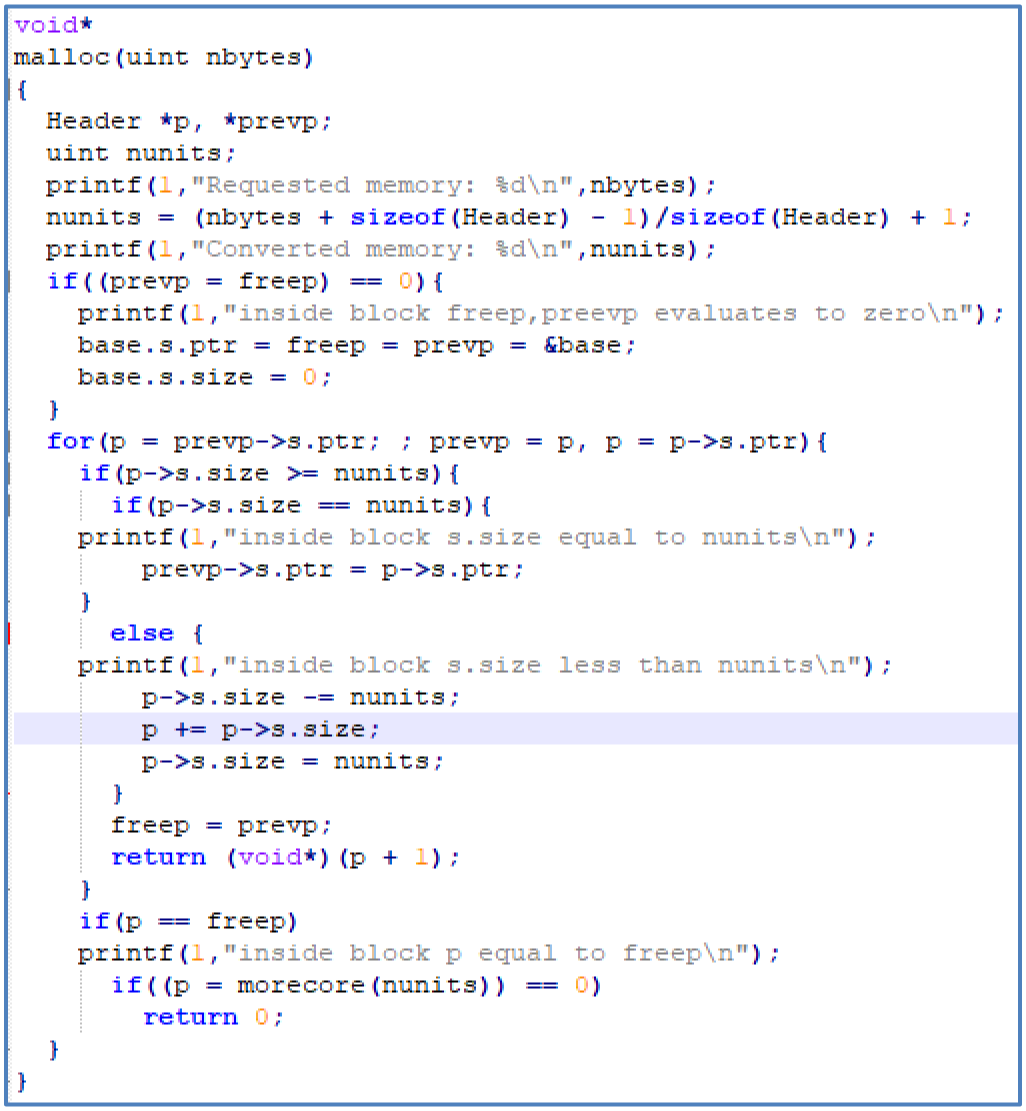
```
void
mallocAndFreeTest(void)
{
  void *m1, *m2;
  int pid, ppid;

  printf(1, "memory test\n");
  ppid = getpid();                    //getting processid
  if((pid = fork()) == 0){            //Creating a process
    m1 = 0;
    //calling malloc function till it cant allocate and assigning it to m1
    while((m2 = malloc(10001)) != 0){
      *(char**)m2 = m1;
      m1 = m2;
    }
    while(m1){        // Freeing allocated memory
      m2 = *(char**)m1;
      free(m1);
      m1 = m2;
    }
    m1 = malloc(1024*20);  // allocating 20KB of memory to m1
    // if m1 is zero then we are writing the failure case memory can't be allocated
    if(m1 == 0){
      printf(1, "couldn't allocate memory?!!\n");
      kill(ppid);  // process is stoped
      exit();
    }
    free(m1); // freeying m1
    printf(1, "memory allocated\n"); // displaying memory allocated
    exit();
  } else {
    wait();
  }
}
```

Case 5:

- Example to illustrate working of the malloc code. And the request to allocate memory was successful.

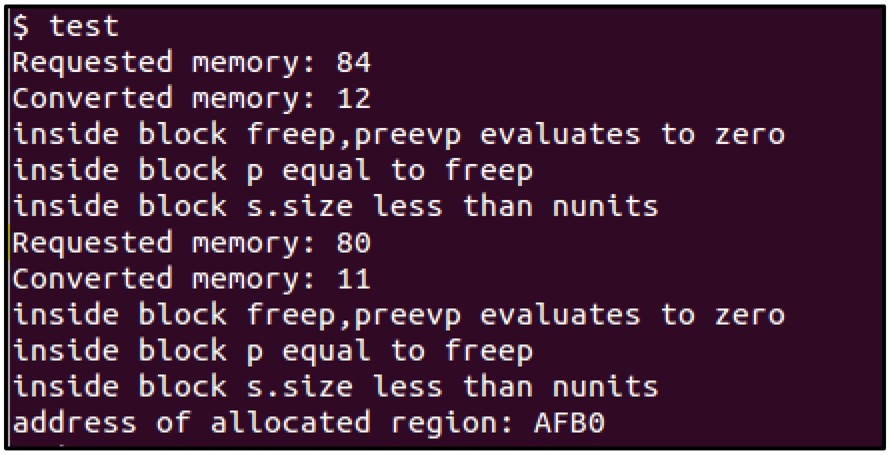


```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "param.h"


#define NULL 0

int
main(int argc, char *argv[])
{


        int* p;
        p= malloc(20 * sizeof(int));
        printf(1,"address of allocated region: %p\n",p);
        return 0;

}
```



```c
void*
malloc(uint nbytes)
{
  Header *p, *prevp;
  uint nunits;
  printf(1,"Requested memory: %d\n",nbytes);
  nunits = (nbytes + sizeof(Header) - 1)/sizeof(Header) + 1;
  printf(1,"Converted memory: %d\n",nunits);
  if((prevp = freep) == 0){
    printf(1,"inside block freep,preevp evaluates to zero\n");
    base.s.ptr = freep = prevp = &base;
    base.s.size = 0;
  }
  for(p = prevp->s.ptr; ; prevp = p, p = p->s.ptr){
    if(p->s.size >= nunits){
      if(p->s.size == nunits){
    printf(1,"inside block s.size equal to nunits\n");
        prevp->s.ptr = p->s.ptr;
    }
      else {
    printf(1,"inside block s.size less than nunits\n");
        p->s.size -= nunits;
        p += p->s.size;
        p->s.size = nunits;
      }
      freep = prevp;
      return (void*)(p + 1);
    }
    if(p == freep)
    printf(1,"inside block p equal to freep\n");
      if((p = morecore(nunits)) == 0)
        return 0;
  }
}
```

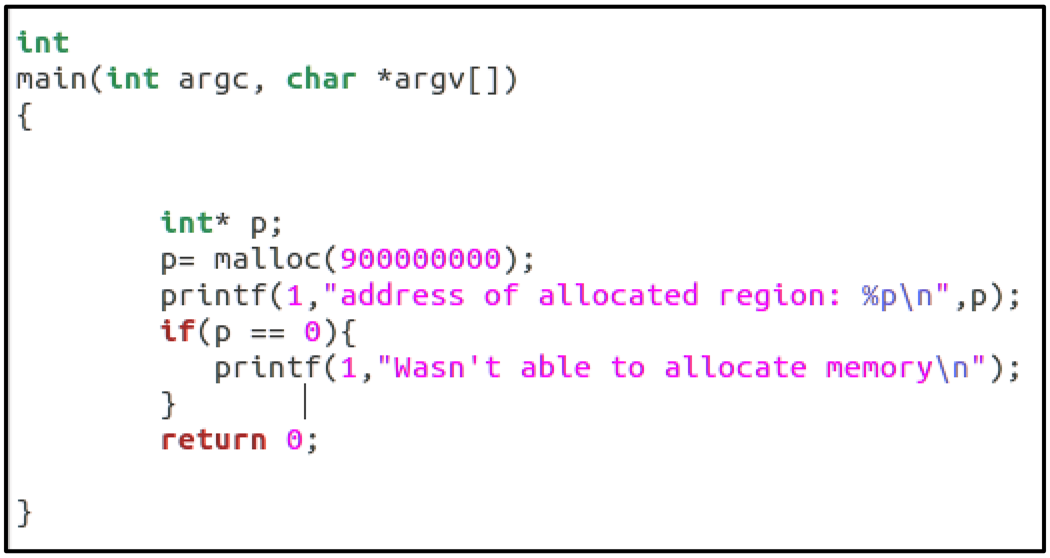
Output:



```
$ test
Requested memory: 84
Converted memory: 12
inside block freep,preevp evaluates to zero
inside block p equal to freep
inside block s.size less than nunits
Requested memory: 80
Converted memory: 11
inside block freep,preevp evaluates to zero
inside block p equal to freep
inside block s.size less than nunits
address of allocated region: AFB0
```

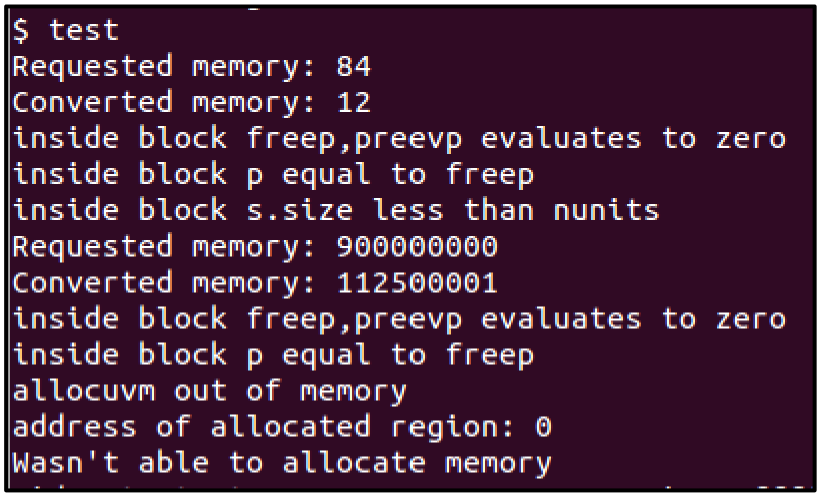
Case 6:

- When the requested memory can’t be allocated. So, it returns zero if the memory couldn’t be allocated.



```
int
main(int argc, char *argv[])
{


        int* p;
        p= malloc(900000000);
        printf(1,"address of allocated region: %p\n",p);
        if(p == 0){
           printf(1,"Wasn't able to allocate memory\n");
        }
        return 0;

}
```

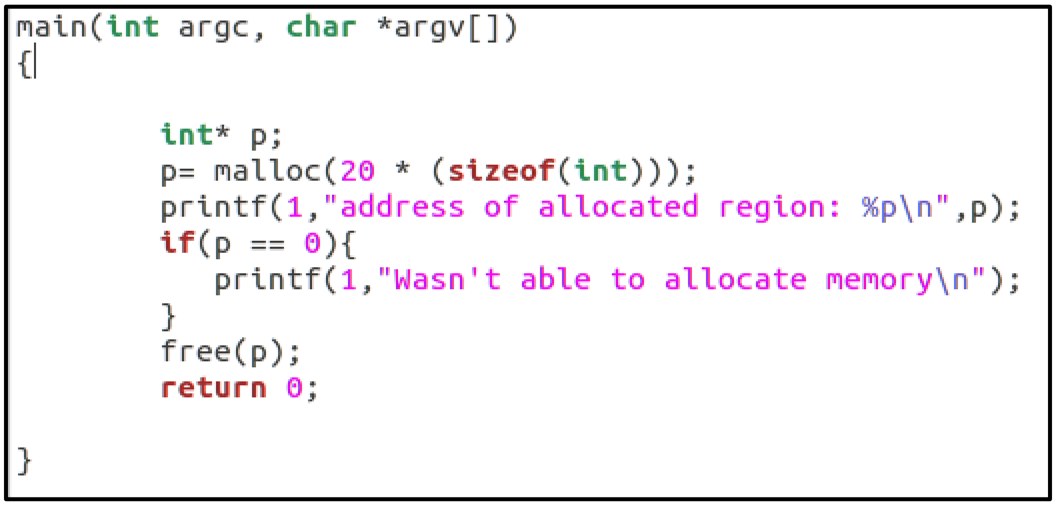
Output:



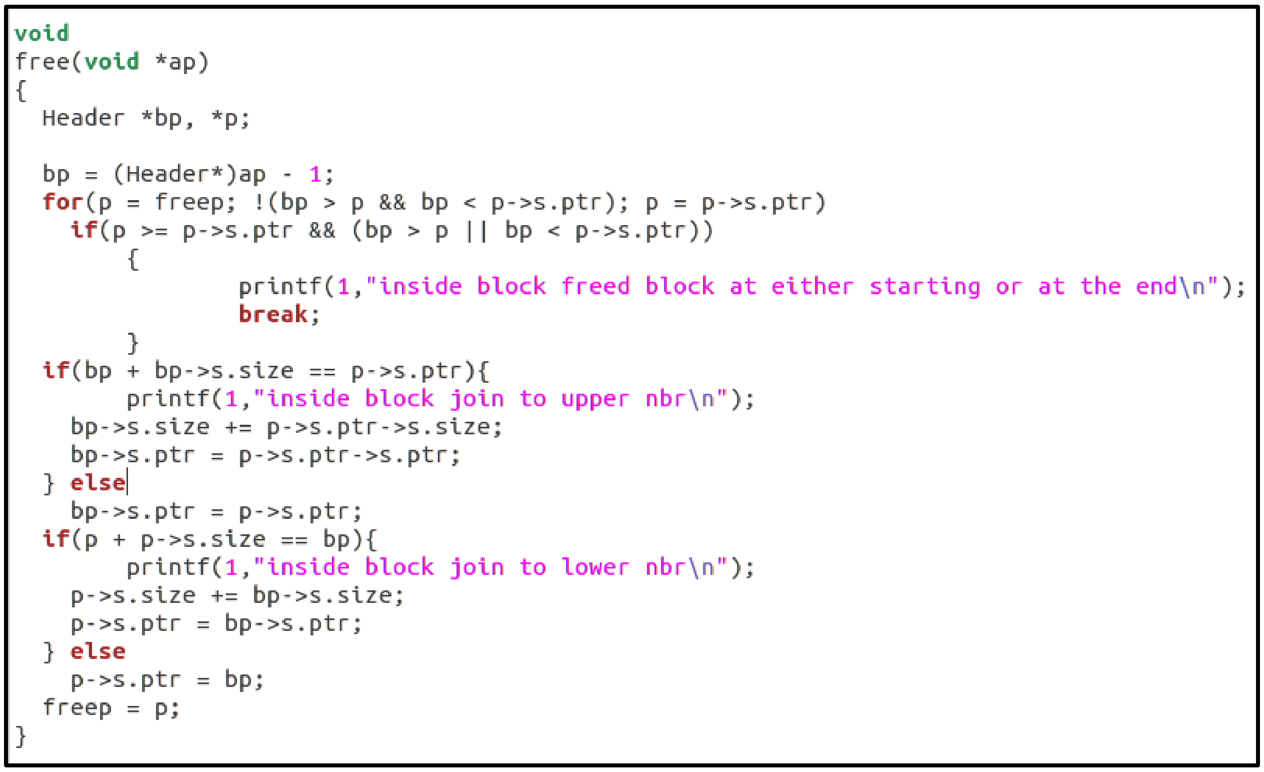
```
$ test
Requested memory: 84
Converted memory: 12
inside block freep,preevp evaluates to zero
inside block p equal to freep
inside block s.size less than nunits
Requested memory: 900000000
Converted memory: 112500001
inside block freep,preevp evaluates to zero
inside block p equal to freep
allocuvm out of memory
address of allocated region: 0
Wasn't able to allocate memory
```

Case 7:

- Example to illustrate working of free function. Given an input of a pointer p. From the output it is obvious that once the memory is freed it has coalesced with the lower block of memory.

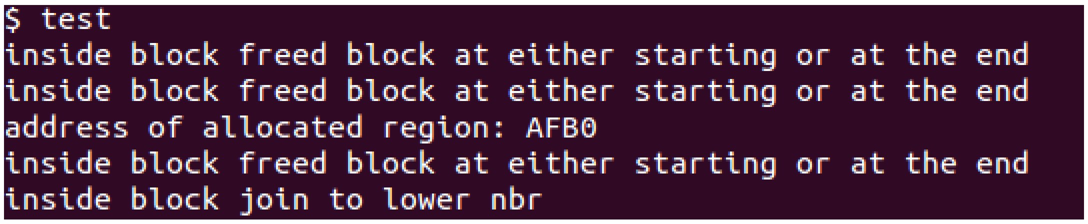


```c
main(int argc, char *argv[])
{

        int* p;
        p= malloc(20 * (sizeof(int)));
        printf(1,"address of allocated region: %p\n",p);
        if(p == 0){
           printf(1,"Wasn't able to allocate memory\n");
        }
        free(p);
        return 0;

}
```



```c
void
free(void *ap)
{
  Header *bp, *p;

  bp = (Header*)ap - 1;
  for(p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
    if(p >= p->s.ptr && (bp > p || bp < p->s.ptr))
        {
                printf(1,"inside block freed block at either starting or at the end\n");
                break;
        }
  if(bp + bp->s.size == p->s.ptr){
        printf(1,"inside block join to upper nbr\n");
    bp->s.size += p->s.ptr->s.size;
    bp->s.ptr = p->s.ptr->s.ptr;
  } else
    bp->s.ptr = p->s.ptr;
  if(p + p->s.size == bp){
        printf(1,"inside block join to lower nbr\n");
    p->s.size += bp->s.size;
    p->s.ptr = bp->s.ptr;
  } else
    p->s.ptr = bp;
  freep = p;
}
```

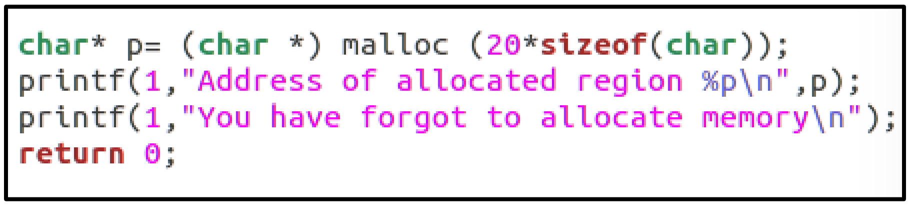
Output:



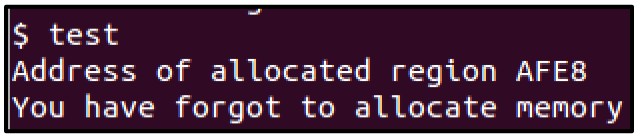
```
$ test
inside block freed block at either starting or at the end
inside block freed block at either starting or at the end
address of allocated region: AFB0
inside block freed block at either starting or at the end
inside block join to lower nbr
```

Case 8:

- **Forgetting to free Memory**



```
char* p= (char *) malloc (20*sizeof(char));
printf(1,"Address of allocated region %p\n",p);
printf(1,"You have forgot to allocate memory\n");
return 0;
```

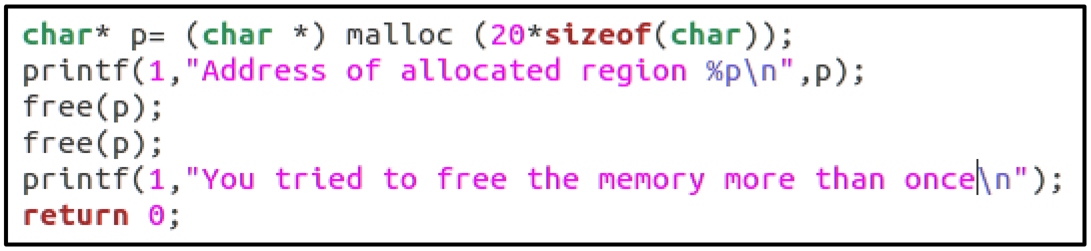


```
$ test
Address of allocated region AFE8
You have forgot to allocate memory
```

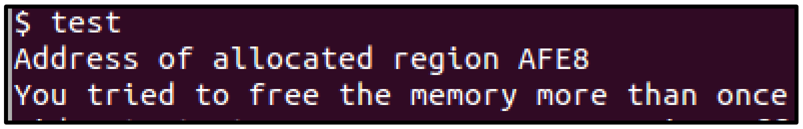
This type of implementation leads to memory leakage and eventually all the memory in the pc will be used up and we need to restart the pc in the worst cases.

Case 9:

- **Freeing memory repeatedly**



```
char* p= (char *) malloc (20*sizeof(char));
printf(1,"Address of allocated region %p\n",p);
free(p);
free(p);
printf(1,"You tried to free the memory more than once\n");
return 0;
```

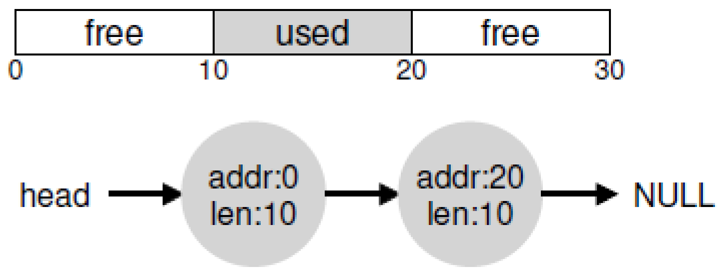


```
$ test
Address of allocated region AFE8
You tried to free the memory more than once
```

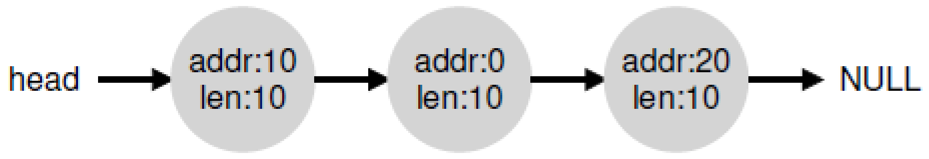
When you do this memory allocation library might get confused and all sort of weird things happen and sometimes result in program crash.

Free Space Management:

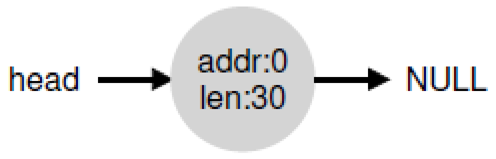
- Free space management issue is handled using free list data structure. It contains all the free blocks of memory in the physical address and allocates a free list based on the requested memory.
- Assume we have 30 byte of heap space and a request is made to give 10 bytes of memory. If we allocate 10 bytes for the requested programs from 10-20 bytes, then if another request comes to allocate more than 10 bytes on memory then the request can't be satisfied. This process of allocating memory is by splitting.



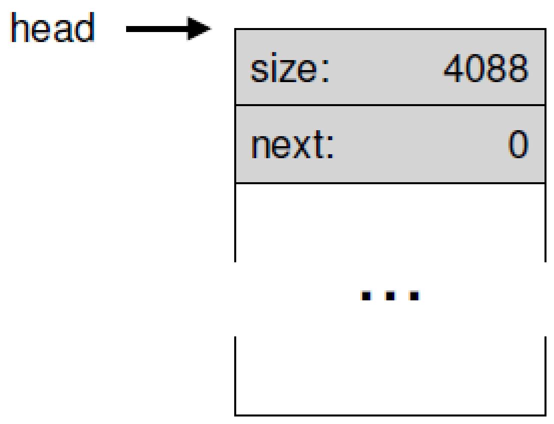
- If a free request comes to free the allocated memory, then the memory is freed, and the block is added back to the free list.



- We can see that even though the address from 0 to 30 is free.
  They are not stored continuously. So, we need to do coalescing by checking the free block on both sides next to the current block.

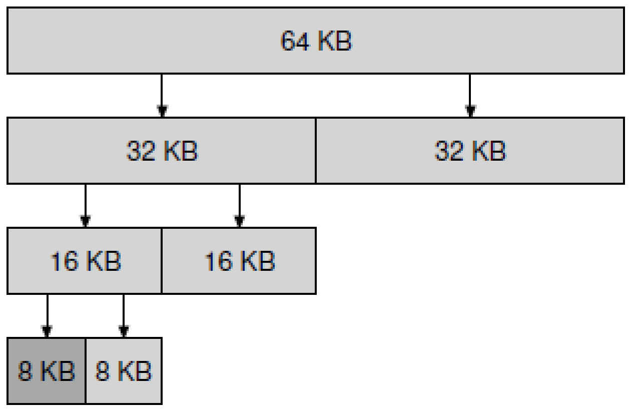


- We will have null in the last list in the free list, to represent the end of the list.
- Diagram of the free list block is shown



- The next in the diagram above has the pointer to the next free block.
- Free called uses this size attribute to remove the allocated memory.
- Since the allocation size varies between request we will have external fragmentation in the physical memory.
- In order to reduce external fragmentation issue there are many techniques, few techniques are listed below.

  1. **Best Fit:** check the block with least size that satisfies the request and allocate from the free list.
  2. **Worst Fit:** Finds the largest size block and returns.
  3. **First Fit:** Finds the first block that satisfies the request from the list of free blocks in the free list and returns.
  4. **Next Fit**: This algorithm keeps an extra pointer to the location within the list where one was pointing last and assigns the requested block of memory.
  5. **Segregated List**: This algorithm knows that certain programs will execute for sure, so it will have free list of that popular request sizes and will allocate memory once they are requested.
  6. **Buddy Allocation Algorithm**: Here free memory is thought of as one big size of 2^N and allocates memory based on the request. This algorithm coalescing is easy it will just merge two blocks of size 8 into a size 16 block. But this algorithm suffers from internal fragmentation.


64 KB
32 KB
32 KB
16 KB
16 KB
8 KB
8 KB