Project# 4 – xv6 Kmalloc & Anonymous mmap Report

He Zhu, hz6627, UT Austin, MSCSO

In this project, I implemented a memory allocator and an anonymous memory mapping system call for xv6. The first part of the project is to implement kmalloc () and kmfree (). And the second part of the project is to implement mmap ().

In the first project, the most important functions' prototypes are displayed as below:

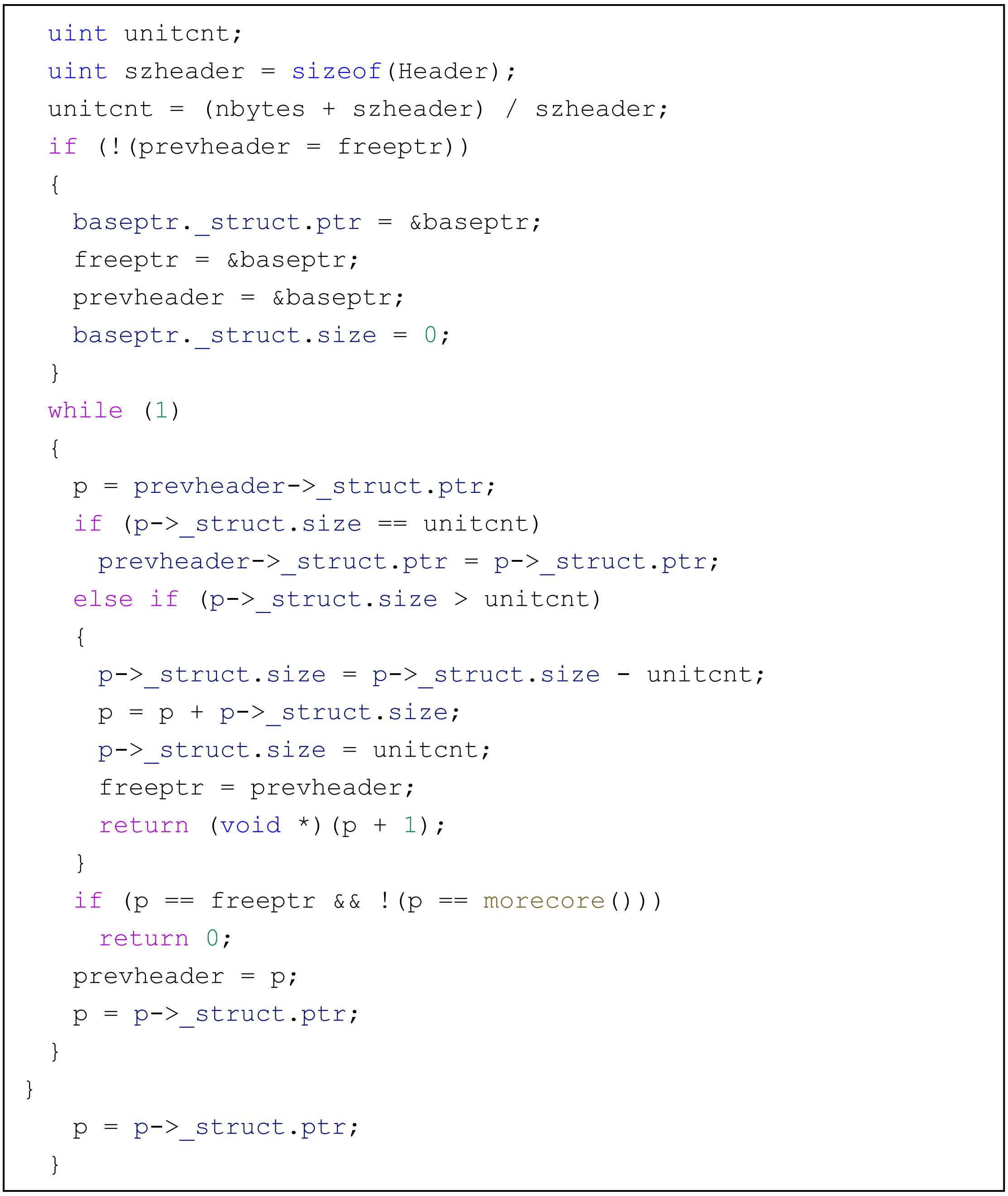
```
//kmalloc.c
void* kmalloc(uint nbytes);
void kmfree(void *addr);
static Header* morecore()
```

The data structure Header has been declared and defined in the *umalloc.c,* which manages the user-space memory. Because xv6 source code does not provide header file including struct Header, so I needed to redefine it in my *kmalloc.c:*

```
//kmalloc.c
union header {
  struct
  {
    uint size;
    union header *ptr;
  } _struct;
  Align x;
};
```

Header is very useful when we come to the memory alignment. According to my view, the Header is declared as union, so it assures that at least 4-bytes memory will be allocated when we work on the 32-bit machine.Now let me discuss kmalloc(). The argument *nbytes* is passed into kmalloc () function, and the kmalloc() uses *nbytes* to determine the size of memory to be allocated. To better explain, please find the code

```
//kmalloc.c
void* kmalloc(uint nbytes)
{
  if (nbytes > BLKSZ)
  {
    panic("Err: over kmalloc() size limit.");
    return 0;
  }
  Header *p;
  Header *prevheader;
```



```
  uint unitcnt;
  uint szheader = sizeof(Header);
  unitcnt = (nbytes + szheader) / szheader;
  if (!(prevheader = freeptr))
  {
    baseptr._struct.ptr = &baseptr;
    freeptr = &baseptr;
    prevheader = &baseptr;
    baseptr._struct.size = 0;
  }
  while (1)
  {
    p = prevheader->_struct.ptr;
    if (p->_struct.size == unitcnt)
      prevheader->_struct.ptr = p->_struct.ptr;
    else if (p->_struct.size > unitcnt)
    {
      p->_struct.size = p->_struct.size - unitcnt;
      p = p + p->_struct.size;
      p->_struct.size = unitcnt;
      freeptr = prevheader;
      return (void *)(p + 1);
    }
    if (p == freeptr && !(p == morecore()))
      return 0;
    prevheader = p;
    p = p->_struct.ptr;
  }
}
    p = p->_struct.ptr;
  }
```

below:
As we can see, the kmalloc() will first check if the memory size to be allocated is greater than *BLKSZ*, here I *#define BLKSZ 4096,* if the memory to be allocated is greater than 4 bytes, than the allocation process will fail and kmalloc() will return 0. We calculated the count of unit we need to allocate the memory by formula:

$$unit\ number = \frac{nbytes + sizeof(Header)}{sizeof(Header)}$$

This is easy to understand. For example, if we pass 8 bytes as nbytes to kmalloc(), the count of units will be (8+4)/4 = 3. Then the Header prevheader will be assigned to freeptr; If the freeptr is not initialized before, all relevant pointer will point to the address of baseptr; Here freeptr and baseptr are both declared as global (static) variables, freeptr means pointer of currently saved free memory block and baseprt is the start of the linked-list of the memory blocks. This design makes the search of free memory starts from free blocks instead of baseptr.

One another important part of the code is in the while-loop, which is nothing but walking though the list. Another noticeable part of the code is *morecore ()* function:

```
//kmalloc.c
static Header *morecore()
{
  char *p;
  Header *hp;
  p = kalloc();

  if (!p)
    return 0;

  hp = (Header *)p;
  hp->_struct.size = BLKSZ;
  kmfree((void *)(hp + 1));
  return freeptr;
}
```

```
//umalloc.c
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;
}
```

Different from the *morecore ()* implemented in umalloc.c which allocated memory by calling *sbrk ()*, the *morecore ()* in the *kmalloc.c ()* called *kalloc ()* to ask for physical memory, and it saves much memory space. For *kmfree ()*, it is almost as the same as the *free ()* in the *umalloc.c,* so I do not show it here to save the page space.

In the second part of the project, I implemented mmap () system call. The prototype of mmap () is

```
void *mmap(void *addr, int length, int prot, int flags, int fd,
           int offset);
int munmap(void *addr, uint length);
```

```
struct mmap_region {
    void *addr;
    int length;
    int prot;
    int offset;
    int fd;
    struct mmap_region *next;}
```

Apart from mmap_region (), we also need to add additional two members to *struct proc* to fully support the mmap () call:

```
uint mmap_size;
struct mmap_region *mmap_begin;
```

Mmap () returns the starting address of the newly mapped memory region. To work on the mmap *()*, we first need to define a data type as above. The member of the data structure is as the same as the arguments passed to the *mmap ()* and a pointer to next region is also included. The most important part of the data structure is *addr* and *length*.

```
void *mmap(void *addr, int length, int prot, int flags, int fd,
int offset)
{
    struct proc *proc = myproc();
    int address = proc->sz;
    if((proc->sz = allocuvm(proc->pgdir, address, address +
length)) == 0)
        return (void*)-1;
    switchuvm(proc);
    struct mmap_region *region = kmalloc(sizeof(struct
    mmap_region));
    region->length = length;
    region->addr = (void*)PGROUNDDOWN((uint)address);
    … …
     struct mmap_region *start = proc->mmap_hd;
     while (start->next!=0)
     {
        start = start->next;
     }
        start->next = region;
    … …
    proc->mmap_sz += 1;
    return (void*)region->addr;
}
```

The *addr* points to the virtual memory address of the mapped region and the *length* is used to save the size of the mapped region in bytes. Please find the code below: (because of the limit of document page number, I only retained key parts here)

Firstly, I allocated a new user level memory block for the mmap () call and switched to the user space. Secondly, apart from simply assigning values to the mmap_region type, I worked on searching a memory space for *mmap ()* in the while loop, a pointer in *mmap_region* type goes through the whole list of mmap regions and add our newly created *mmap_region* pointer to the end of list, thus creating a complete list to manage the memory allocated by the mmap ();

The project is ended up with the munmap () system call, which will destroy the

```
int munmap(void *addr, uint length)
{
    struct proc *proc = myproc();
    int procsize = proc->sz;
    struct mmap_region *begin = proc->mmap_begin;
    struct mmap_region *it;
    int is_found = 0;

    if(begin->addr == addr && begin->length == length)
        … …
    else
    {
        while(begin->next)
        {
            if(start->next->addr == addr && start->next->length
== length)
            {
                begin->next = begin->next->next;
                is_found = 1;
                break;
            }
            start = begin->next;
        }
    }

    if(is_found)
    {
        if((proc->sz = deallocuvm(proc->pgdir, procsize,
procsize - length)) == 0)
        {
            free_mmap_regions(proc->mmap_begin);
            return -1;
        }

        switchuvm(proc);
        proc->mmap_size -= 1;
        return 0;
    }
    free_mmap_regions(proc->mmap_begin);
    return -1;
}
```

mappings for the specified address range. Above I attached the most important code segments in this function. Same as the mmap (), munmap() also calls the myproc () to get the current process context; ; Then we will check if the size and the address of the to be deleted mmapped region same as those save in the current process context, if so, we locate it; if not, we again go through the linked list of the mapped regions until we find the memory region we are looking for. At last, we need to deallocate (*deallocateuvm ()*) and free (*kmfree ()*) the mapped memory space and mapped regions, and we switch back to the user context.