تمرین سری اول آزمایشگاه سیستم‌های عامل
(نخ‌ها و پشته)

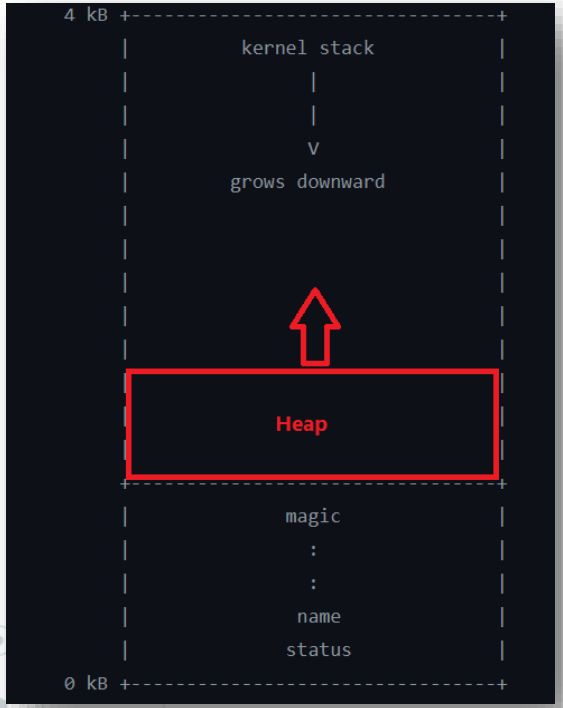
استاد درس
دکتر شهاب‌الدّین نبوی

اعضای گروه
امیر حلاجی بیدگلی
حامد خادمی خالدی
محمد خدام
متین زیودار

بهار ۱۴۰۰

۱. پیش‌گفتار

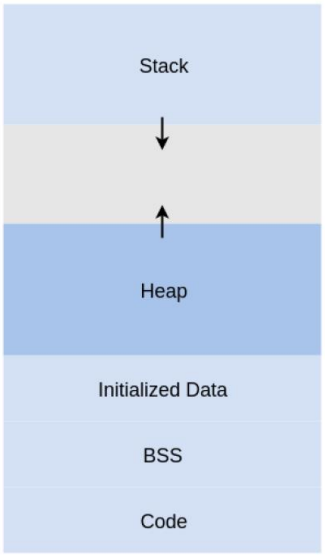
هدف پروژه

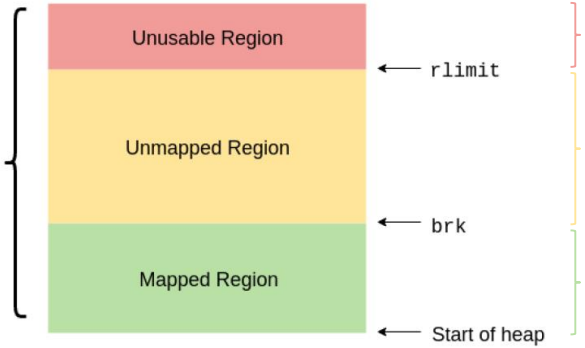


هدف از این پروژه پیاده سازی حافظه پویا برای هر ترد در سیستم عامل **Pintos** می‌باشد.
هر ترد یا پروسس در پینتوس حافظه اختصاصی خود را دارد که شکل آن به صورت زیر است (هدف ما پیاده سازی بخش Heap است).
برای پیاده سازی Heap نیاز به توابع مختلفی نیاز است که در ادامه به بررسی کد تک‌تک آن‌ها پرداخته و آن‌ها را شرح می‌دهیم.

۲. Heap

ساختار Heap



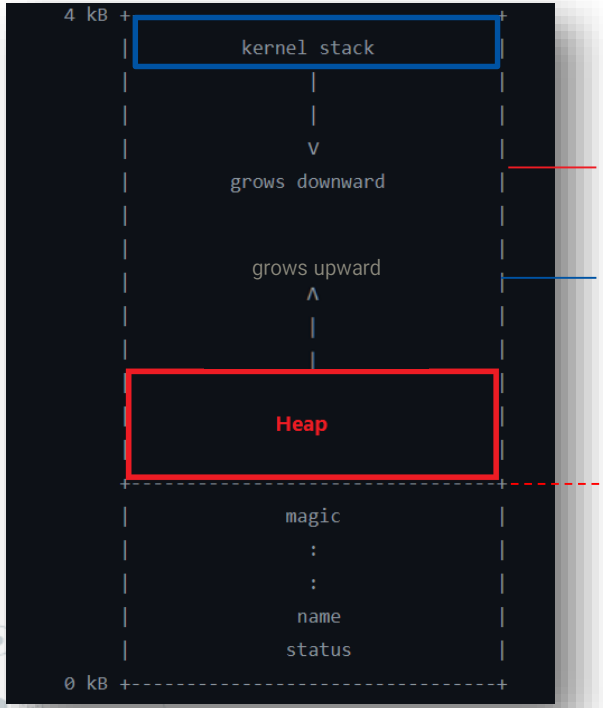


Not accessible space,
Bound Limit for Heap (Not implemented in this project)

Accessible space but needs OS privilege

Accessible space

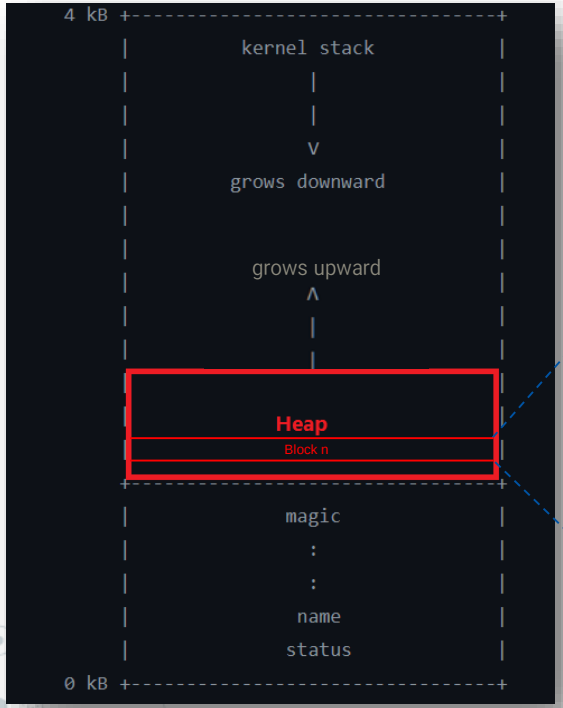
چالش‌های Heap



۱. امکان تداخل با استک

۲. آدرس شروع از آدرس صفر حافظه نیست.

ساختمان داده Heap



۳. تشریح توابع

تابع extend_heap

```
s_block_ptr extend_heap (s_block_ptr last , size_t s){
    void* p=sbrk(s+sizeof(s_block));

    if(!(p==(void *)-1)){

        s_block_ptr newBlock = (s_block_ptr) p;
        if(last){
            last->next=newBlock;

        }else{
            HeadPtr=newBlock;
        }
        newBlock->prev=last;
        newBlock->next=NULL;
        newBlock->size=s;
        newBlock->block=p+sizeof(s_block);
        //last->ptr=p;
        newBlock->free=0;

        return newBlock;
    }
    //Return Null on error
    return NULL;
}
```

این تابع در صورت امکان یک بلاک به انتهای heap اضافه می‌کند و در غیر این صورت null بر می‌گرداند.

برای پیاده سازی این تابع از سیستم کال sbrk استفاده می‌شود تا چک کند که آیا می‌توان heap را گسترش داد یا خیر.

ورودی:

۱- اشاره‌گر به ابتدای بلوک جدید

۲- سایز بلوک جدید (بخش دیتا)

خروجی:

اشاره‌گری به ابتدای بلوک جدید یا null

تابع get_block

این تابع یک آدرس گرفته و اگر این آدرس به ابتدای یک بلاک اشاره کند آن بلاک را برمی‌گرداند و در غیر این صورت nullباز می‌گرداند.

```
s_block_ptr get_block (void *p){
    s_block_ptr head=HeadPtr;

    while(head){
        if(head->block==p){
            return head;
        }
        head=head->next;
    }
    //null if we cant find it.
    return NULL;

}
```

ورودی:
۱- اشاره‌گر به ابتدای بلوک موردنظر
خروجی:
اشاره‌گری به ابتدای بلوکِ موردنظرِ یافت‌شده یا null

تابع split_block

```
void split_block (s_block_ptr b, size_t s){
    if(b && s >= sizeof(void *)){
        if(  b->size - s  >= sizeof(s_block) + sizeof(void*) ){
            s_block_ptr newP = (s_block_ptr) (b->block + s);
            newP->next = b->next;
            (newP->next)->prev=newP;
            b->next=newP;
            newP->prev=b;
            newP->size=b->size - s - sizeof(s_block);
            b->size = s;
            newP->block= b->block + s + sizeof(s_block);

            mm_free(newP->block);

        }
    }
}
```

یک اشاره گر به یک بلاک و یک سایز از ورودی گرفته و بلاک مد نظر را به اندازه آن تقسیم می‌کند.
لازم اندازه این سایز یک حداقلی داشته باشد و همچنان از اندازه بلاک بزرگتر نباشد و بعد از چک کردن به سراغ درست کردن ارتباطات بین بلاک‌های جدید می‌رویم.

ورودی:
۱- اشاره‌گر به ابتدای بلوک جدید
۲- سایز انتخابی بلوک جدید جهت split کردن
خروجی:
ندارد.

تابع fusion

یک اشاره‌گر به یک بلاک را گرفته و اگر بتوان آن را با هریک از همسایه‌ها ادغام کرد آن را ادغام می‌کند (لازم است ارتباط بین بلاک ها - next and prevنیز درست شوند)

```
s_block_ptr fusion(s_block_ptr b){
    if( (b->next)->free ==1 ){
        b->size=b->size+sizeof(s_block)+(b->next)->size;
        b->next=(b->next)->next;
        (b->next)->prev=b;
    }

    if( (b->prev)->free ==1 ){

        (b->prev)->next=b->next;
        (b->next)->prev=b->prev;
        (b->prev)->free=b->free;
        (b->prev)->size=(b->prev)->size + sizeof(s_block) + b->size;

        return b->prev;
    }

    return b;
}
```

ورودی:
۱- اشاره‌گر به ابتدای بلوک
خروجی:
اشاره‌گری به ابتدای بلوک جدید extend شده یا همان آدرس ابتدای بلوک اولیه

تابع mem_copy

این تابع، دو آرگومان دارد که محتویات بلاک قدیمی را در بلاک جدید، کپی می‌کند. در زمان‌هایی که بخواهیم از حافظه بزرگتری استفاده کنیم و آن را نداشته باشیم، پس از ایجاد کردن حافظه جدید، محتویات آن را با استفاده از این تابع، کپی می‌کنیم.

ورودی:
۱- اشاره‌گر به ابتدای بلوک اول
۲- اشاره‌گر به ابتدای بلوک دوم
خروجی:
اشاره‌گری به بخش دیتای بلوک جدید
یا null

```
void* mem_copy(s_block_ptr oldB, s_block_ptr newB){
    if( oldB && newB){
        char * oStart= (char *) oldB->block;
        char * nStart=(char *) newB->block;
        int i;
        for(i=0;i<oldB->size;i++){
            *(nStart + i)=*(oStart + i);
        }
        return newB->block;
    }
    return NULL;
}
```

تابع mm_realloc

```
void* mm_realloc(void* ptr, size_t size){
        s_block_ptr curr=get_block(ptr);

        if(curr){
            if(size > curr->size){
                void* p=mm_malloc(size);
                s_block_ptr newP=get_block(p);
                if(newP){
                    p=mem_copy(curr,newP);
                    mm_free(curr->block);
                    return p;
                }
            }else if(size < curr->size){
                split_block(curr,size);
                return curr->block;
            }else{
                return curr->block;
            }
        }
        return NULL;
}
```

این تابع، دو پارامتر دارد که یکی از آن‌ها، اشاره‌گری به یک بلاک و دیگری، اندازه بلاک است. کاری که می‌کند این است که سایز بلاک را به اندازه سایز دلخواه تغییر بدهد. یک نمونه استفاده از تابع memcpy، میتواند این باشد که ابتدا با تابع realloc، یک حافظه بزرگتر بخواهیم و سپس memcpyکنیم.

ورودی:
۱- اشاره‌گر به ابتدای یک بلوک
۲- اندازه جدید بلاک
خروجی:
اشاره‌گری به دیتای بلوک جدید
تغییراندازه داده‌شده یا null

تابع mm_malloc

این تابع در ورودی یک سایز می‌گیرد و به دنبال یافتن اولین خانه در حافظه است که خالی باشد و حداقل اندازه مورد نظر را نیز داشته باشد.

ورودی:
۲- سایز بلوک
خروجی:
اشاره‌گری به بخش دیتای بلوک جدید یا بلوک اولیه

```
void* mm_malloc(size_t size){

    s_block_ptr head=HeadPtr;
    s_block_ptr prev=NULL;

    while(head){
        if(head->free==1 && head->size >=size){
            mm_realloc(head->block,size);
            head->free=0;
            return head->block;

        }else{
            prev=head;
            head=head->next;
        }
    }

    head=extend_heap(prev,size);
    if(!head){
        exit(EXIT_FAILURE);
    }
    return head->block;

}
```

تابع mm_free

```
void mm_free(void* ptr){
    s_block_ptr curr=get_block(ptr);
    if(curr){
        if(curr->next==NULL){
            if(curr->prev){

                (curr->prev)->next=NULL;
            }else{
                HeadPtr=NULL;
            }
            sbrk(-(curr->size + sizeof(s_block)));
        }else{
            curr->free=1;
            fusion(curr);
        }
    }
}
```

همانطور که از اسم این تابع مشخص است، این تابع برا آزادسازی است. اینگونه است که یک اشاره‌گر در ورودی می‌گیرد و بلاکی که این اشاره‌گر به آن اشاره می‌کند را، آزادسازی می‌کند. لازم به ذکر است علاوه بر اینکه اشاره‌گر های nextو prevرا nullمی‌کنیم، باید تابع fusionرا نیز استفاده کنیم که نسبت به ادغام دو فضایی که اکنون از هم جدا هستند، اقدام کند.

ورودی:
۱- اشاره‌گر به یک بلوک
خروجی:
- - -

پایان

با تشکر از توجه شما

منابع:

https://inst.eecs.berkeley.edu/~cs162/fa19/static/hw/hw5.pdf

https://github.com/thecuongthehieu/CS-162-Operating-Systems-and-System-Programming-Homework