We will use malloc() to allocate 3 continuously memory blocks: Block1, block2, block 3, each with size of 4088bytes

1. First, we want to apply an error case for the uv2p() system call, since virtual address of block1 is the first entry of page table, we use (virtual address of block1 + 4096), which virtual address is in another page and hasn't been allocated yet, the uv2p() will return -1.

2. The below step will demonstrate the malloc() and free() split and coalescing charactors.
   - Free block2, and physical memory of block2 is free
   - Then malloc(100) for block4 which size is smaller than block2 size, the malloc() will split block2 memory, allocate block4 with same physical address of block2
   - Then free block4, free() will coalesce the block4 and the rest of block 2 memory
   - At last, we allocate block5 with size of 4088, malloc() give the block2 memory which is just coalesced by free(), to block5
3. Test code:

```
int
main(void)
{

//char uinput[20];
char* block1=malloc(4088);
char* block2=malloc(4088);
char* block3=malloc(4088);
char* block4;
char* block5;
printf(1,"\n virtual address not existed in page table is: %p ",block1+4096);
if(uv2p(block1+4096)<0)
    printf(1,"input virtual address not exist in page table");

printf(1,"\nblock1 virtual address is: %p ",block1);
if(uv2p(block1)<0)
    printf(1,"input virtual address not exist in page table");

printf(1,"\nblock2 virtual address is: %p ",block2);
if(uv2p(block2)<0)
    printf(1,"input virtual address not exist in page table");
printf(1,"\nblock3 virtual address is: %p ",block3);
if(uv2p(block3)<0)
    printf(1,"input virtual address not exist in page table");

free(block2);
block4=malloc(100);
printf(1,"\nfree(block2) and block4=malloc(100)");

printf(1,"\nblock4 virtual address is: %p ",block4);
if(uv2p(block4)<0)
    printf(1,"input virtual address not exist in page table");

free(block4);
free(block3);
block5=malloc(8000);
printf(1,"\nfree(block3) free(block4) and block5=malloc(8000)");
printf(1,"\nblock5 virtual address is: %p ",block5);
if(uv2p(block5)<0)
    printf(1,"input virtual address not exist in page table");
free(block1);
free(block5)
exit();
}
```

Call Malloc() process ------------page2—page9

Call Free() process ------------ page10

1. User program will include user.h file, which define function symbol malloc(uint) and free(void *) when user call malloc() and free(), program will execute these functions in file umalloc.c

```
typedef long Align;

union header {
  struct {
    union header *ptr;
    uint size;
  } s;
  Align x;
};
typedef union header Header;
```

```
static Header base;
static Header *freep;
```

```
 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;
  }
```

the malloc function first calculate given nbytes into nuits, for the first time our program require memory, the free list has not created yet, freep is 0, this block will run, initialize base.s.ptr, prevp and freep pointer all to base address, set base.s.size to zero. So at first malloc,

2. Malloc()continuously loop in freep list, if find a node space = nunits, occupy this node, and remove this node from freep list. If space>nunits, split this node space, if after iterate the whole list, space<nunits, call morecore() to allocate more physical memory of size nunits for current process;

```
  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;
  }
```

3. In umalloc.c, morecore will allocate memory size as given parameter nu, first judge if the nu is less than PAGE(4096) , if less, allocate a whole page. Morecore will call system call sbrk(), pass the number of bytes that will be allocated.

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

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

PAGE is a global symbol defined in file para.h

```
#define NPROC        64  // maximum number of processes
#define PAGE       4096  // granularity of user-space memory allocation
#define KSTACKSIZE PAGE  // size of per-process kernel stack
#define NCPU          8  // maximum number of CPUs
#define NOFILE       16  // open files per process
```

4. for homework 2 already demonstrates the system call process, we here will jump to the actual execute code in file sysproc.c, function sys_sbrk()

```
int
sys_sbrk(void)
{
  int addr;
  int n;

  if(argint(0, &n) < 0)
    return -1;
  addr = cp->sz;
  if(growproc(n) < 0)
```

```
    return -1;
  return addr;
}
```

sys_sbrk will call argint() in file syscall.c to get the argument of the system call sbrk(argument) given by libc.

```
int
argint(int n, int *ip)
{
  return fetchint(cp, cp->tf->esp + 4 + 4*n, ip);
}
```

```
int
fetchint(struct proc *p, uint addr, int *ip)
{
  if(addr >= p->sz || addr+4 > p->sz)
    return -1;
  *ip = *(int*)(p->mem + addr);
  return 0;
}
```

5. In file sysproc.c, after function sys_sbrk() call argint() to get the argument n, which is number of bytes required to allocate, it will call growproc() in file proc.c

```
// Grow current process's memory by n bytes.
// Return 0 on success, -1 on failure.
int
// Grow current process's memory by n bytes.
// Return 0 on success, -1 on failure.
int
growproc(int n)
{
  uint sz;
  struct proc *curproc = myproc();

  sz = curproc->sz;
  if(n > 0){
    if((sz = allocuvm(curproc->pgdir, sz, sz + n)) == 0)
      return -1;
  } else if(n < 0){
    if((sz = deallocuvm(curproc->pgdir, sz, sz + n)) == 0)
```

```
      return -1;
  }
  curproc->sz = sz;
  switchuvm(curproc);
  return 0;
}
```

growproc() first call myproc() to get current user program pointer
Then call allocuvm() and deallocuvm() to allocate a bigger or smaller memory than current program .
Allocuvm() first if the new size is bigger than old size, if not, just return the old size.
If is bigger, round old size up to whole page sizes, call kalloc() to allocate page sizes physical memory, set memory value to 0, call mappages() to set the new allocated memory physical address to the page table,
if failed, call deallocateuvm() to ,call kfree() to free the physical memory, return 0
if success, return the new allocated memory virtual address.

```
/ Allocate page tables and physical memory to grow process from oldsz to
// newsz, which need not be page aligned.  Returns new size or 0 on error.
int
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
  char *mem;
  uint a;

  if(newsz >= KERNBASE)
    return 0;
  if(newsz < oldsz)
    return oldsz;

  a = PGROUNDUP(oldsz);
  for(; a < newsz; a += PGSIZE){
    mem = kalloc();
    if(mem == 0){
      cprintf("allocuvm out of memory\n");
      deallocuvm(pgdir, newsz, oldsz);
      return 0;
    }
    memset(mem, 0, PGSIZE);
    if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
      cprintf("allocuvm out of memory (2)\n");
      deallocuvm(pgdir, newsz, oldsz);
      kfree(mem);
      return 0;
    }
  }
  return newsz;
}
```

in file kalloc.c, which charge for the physical memory allocation. Give kalloc() parameter “cp->sz+n”, the sum of current user program memory size PLUS additional new n bytes.

```
// Allocate n bytes of physical memory.
// Returns a kernel-segment pointer.
// Returns 0 if the memory cannot be allocated.
char*
kalloc(int n)
{
  char *p;
  struct run *r, **rp;

  if(n % PAGE || n <= 0)
    panic("kalloc");

  acquire(&kmem.lock);
  for(rp=&kmem.freelist; (r=*rp) != 0; rp=&r->next){
    if(r->len == n){
      *rp = r->next;
      release(&kmem.lock);
      return (char*)r;
    }
    if(r->len > n){
      r->len -= n;
      p = (char*)r + r->len;
      release(&kmem.lock);
      return p;
    }
  }
  release(&kmem.lock);
  cprintf("kalloc: out of memory\n");
  return 0;
}
```

for each virtual address and physical address of page pieces of given memory, mappages() will call walkpgdir() to go though the page table for the given virtual address, if miss, walkpgdir() will create a new entry, return the new entry pointer to mappages(); if match, walkpgdir() return the found entry pointer.

Mappages() write the physica address to the returned page table entry, mark the entry flag.

```
// Create PTEs for virtual addresses starting at va that refer to
// physical addresses starting at pa. va and size might not
// be page-aligned.
static int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

```
// Return the address of the PTE in page table pgdir
// that corresponds to virtual address va.  If alloc!=0,
// create any required page table pages.
```

```c
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  pde = &pgdir[PDX(va)];
  if(*pde & PTE_P){
    pgtab = (pte_t*)P2V(PTE_ADDR(*pde));
  } else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
      return 0;
    // Make sure all those PTE_P bits are zero.
    memset(pgtab, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;
  }
  return &pgtab[PTX(va)];
}
```

Free()

1. When user program call free(), program will execute free() in file umalloc.c
2. ap is virtual address pointer to the block that will be free, free() first cast ap to header type, minus 1 so bp will be the pointer of the header address of this block, loop the free list, break when bp is between two node p and p->s.ptr in the free list
3. if block of bp just before the block pointed by p->s.ptr, there is no gap between these two block, coalescing these two block by setting bp->s.ptr to p->s.ptr->s.ptr else situation is end of block bp can't reach block p->s.ptr, then set bp->s.ptr pointing to p->s.ptr
4. Then consider the bp block and p block which is before bp:

   If there is no gap between these two block, coalescing these two block, set p->s.size = sum of these two block, set p pointing to bp->s.ptr

   Else situation is there is gap between block p and bp, then set p-s.ptr pointing to bp

```
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;
}
```