```
(gdb)
main () at test.c:11
11          ptr=(int*)malloc(sizeof(int));
(gdb)
malloc (nbytes=4) at umalloc.c:69
69          nunits = (nbytes + sizeof(Header) - 1)/sizeof(Header) + 1;
(gdb)
70          if((prevp = freep) == 0){
(gdb)
69          nunits = (nbytes + sizeof(Header) - 1)/sizeof(Header) + 1;
(gdb)
70          if((prevp = freep) == 0){
(gdb)
71            base.s.ptr = freep = prevp = &base;
(gdb)
72            base.s.size = 0;
(gdb)
86            if(p == freep)
(gdb)
87              if((p = morecore(nunits)) == 0)
(gdb)
morecore (nu=4096) at umalloc.c:54
54          p = sbrk(nu * sizeof(Header));
(gdb)
sbrk () at usys.S:29
29        SYSCALL(sbrk)
```

Figure 1: GDB steps

From the GDB tracing we find that malloc returns to sbrk. For the purpose of the exercise, the parsing portion will be ignored to focus on the concepts of malloc.

```c
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;
  }
}
```
Figure 2: umalloc.c malloc

To being, the user calls on the function malloc from umalloc.c. It is evident that given valid conditionsit will adjust the pointers and size of the header. Once adjusted and the new header is valid, it will call on morecore.

```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;
}
```
Figure 3: umalloc.c morecore

morecore then gives the new unit size (minimum 4K for page size) to sbrk.

```
#define SYSCALL(name) \
  .globl name; \
  name: \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL; \
    ret
```
```
SYSCALL(sbrk)
```
Figure 4: usys.S user sbrk

```c
#define SYS_sbrk   12
```
Figure 5: syscall.h SYS_sbrk define

The function sbrk can be found from the usys.S defined library of functions. This utilizes syscall.h to return the syscall number, resulting in the appropriate arguments and registers before performing the trap.

```c
// These are arbitrarily chosen, but with care not to overlap
// processor defined exceptions or interrupt vectors.
#define T_SYSCALL       64      // system call
```
Figure 6: traps.h T_SYSCALL define

The pushed arguments are then compared to traps.h definitions to determine which type it is, in this case a T_SYSCALL. At this point privilege/level is changed as the trap function changes from user to kernel.

```c
// Bootstrap processor starts running C code here.
// Allocate a real stack and switch to it, first
// doing some setup required for memory allocator to work.
int
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  tvinit();        // trap vectors
```

Figure 7: main.c main

```c
void
tvinit(void)
{
  int i;

  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
  SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);

  initlock(&tickslock, "time");
}
```

Figure 8: trap.c tvinit

```c
// Set up a normal interrupt/trap gate descriptor.
// - istrap: 1 for a trap (= exception) gate, 0 for an interrupt gate.
//   interrupt gate clears FL_IF, trap gate leaves FL_IF alone
// - sel: Code segment selector for interrupt/trap handler
// - off: Offset in code segment for interrupt/trap handler
// - dpl: Descriptor Privilege Level -
//        the privilege level required for software to invoke
//        this interrupt/trap gate explicitly using an int instruction.
#define SETGATE(gate, istrap, sel, off, d)                \
{                                                         \
  (gate).off_15_0 = (uint)(off) & 0xffff;                \
  (gate).cs = (sel);                                      \
  (gate).args = 0;                                        \
  (gate).rsv1 = 0;                                        \
  (gate).type = (istrap) ? STS_TG32 : STS_IG32;           \
  (gate).s = 0;                                           \
  (gate).dpl = (d);                                       \
  (gate).p = 1;                                           \
  (gate).off_31_16 = (uint)(off) >> 16;                  \
}
```
Figure 9: mmu.h SETGATE

It is important to note that the trap table has been initialized during the boot process of the system, which allows it so that any point after boot to execute the defined trap functions. This is done with defining the offsets of the trap vectors.

```c
// Common CPU setup code.
static void
mpmain(void)
{
  cprintf("cpu%d: starting %d\n", cpuid(), cpuid());
  idtinit();       // load idt register
  xchg(&(mycpu()->started), 1); // tell startothers() we're up
  scheduler();     // start running processes
}
```
Figure 10: main.c mpmain

```c
void
idtinit(void)
{
  lidt(idt, sizeof(idt));
}
```

Figure 11: trap.c idtinit

```c
static inline void
lidt(struct gatedesc *p, int size)
{
  volatile ushort pd[3];

  pd[0] = size-1;
  pd[1] = (uint)p;
  pd[2] = (uint)p >> 16;

  asm volatile("lidt (%0)" : : "r" (pd));
}
```

Figure 12: x86.h lidt

The previous set of figures lists how the processor itself (hardware) accesses the table from memory, which in turn allows for some work to be done at the hardware level. While this sequence of trap table tracing was not necessary, it explains priori the access of the trap from memory (and sequentially this occurs before user call as it is set during boot). At this point the trap table is set up, and the tracing of sbrk (and malloc) may continue.

```asm
.globl vector64
vector64:
  pushl $0
  pushl $64
  jmp alltraps
```

Figure 13: vectors.S global vector

```
  # vectors.S sends all traps here.
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  # Set up data segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es

  # Call trap(tf), where tf=%esp
  pushl %esp
  call trap
  addl $4, %esp
```

Figure 14: trapasm.S alltraps

Continuing from sbrk when the trap is called, the first thing performed is a push of the trap number followed by a saving of registers to the stack. Afterwards, these registers are adjusted to access kernel memory for trap handling.

```
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(myproc()->killed)
      exit();
    myproc()->tf = tf;
    syscall();
    if(myproc()->killed)
      exit();
    return;
  }
```

Figure 15: trap.c trap

The trap number read 64, or T_SYSCALL, thus focusing on that segment of the trap function. It is at this point where the table of system calls is referred to for the statements to perform.

```c
void
syscall(void)
{
  int num;
  struct proc *curproc = myproc();

  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
```

Figure 16: syscall.c syscall

```c
static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_sbrk]    sys_sbrk,
```

Figure 17: syscall.c syscalls[]

In the function it is found that eax is referred to, better known as the syscall number (12). In this case, sys_sbrk is called and its return value is set back to eax.

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

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

Figure 18: sysproc.c sys_sbrk

Now beginning with the syscall for sbrk, the sub-function argomt is done to validate the argument for malloc.

```c
// Fetch the nth 32-bit system call argument.
int
argint(int n, int *ip)
{
  return fetchint((myproc()->tf->esp) + 4 + 4*n, ip);
}
```

Figure 19: syscall.c argint

```c
// Fetch the int at addr from the current process.
int
fetchint(uint addr, int *ip)
{
  struct proc *curproc = myproc();

  if(addr >= curproc->sz || addr+4 > curproc->sz)
    return -1;
  *ip = *(int*)(addr);
  return 0;
}
```

Figure 20: syscall.c fetchint

As provided with in-code documentation, it pulls the process state (myproc is the structure) and fetches the nth 32-bit syscall argument. In fetchint, it validates the address for bounds and size before returning (rewriting the pointer). Now exploring the myproc:

```c
// Disable interrupts so that we are not rescheduled
// while reading proc from the cpu structure
struct proc*
myproc(void) {
  struct cpu *c;
  struct proc *p;
  pushcli();
  c = mycpu();
  p = c->proc;
  popcli();
  return p;
}
```

Figure 21: proc.c myproc

```c
// Pushcli/popcli are like cli/sti except that they are matched:
// it takes two popcli to undo two pushcli.  Also, if interrupts
// are off, then pushcli, popcli leaves them off.

void
pushcli(void)
{
  int eflags;

  eflags = readeflags();
  cli();
  if(mycpu()->ncli == 0)
    mycpu()->intena = eflags & FL_IF;
  mycpu()->ncli += 1;
}
```

Figure 22: spinlock.c pushcli

```c
static inline uint
readeflags(void)
{
  uint eflags;
  asm volatile("pushfl; popl %0" : "=r" (eflags));
  return eflags;
}
```

Figure 23: x86.h readeflags

```c
static inline void
cli(void)
{
  asm volatile("cli");
}
```

Figure 24: x86.h cli

```c
// Must be called with interrupts disabled to avoid the caller being
// rescheduled between reading lapicid and running through the loop.
struct cpu*
mycpu(void)
{
  int apicid, i;

  if(readeflags()&FL_IF)
    panic("mycpu called with interrupts enabled\n");

  apicid = lapicid();
  // APIC IDs are not guaranteed to be contiguous. Maybe we should have
  // a reverse map, or reserve a register to store &cpus[i].
  for (i = 0; i < ncpu; ++i) {
    if (cpus[i].apicid == apicid)
      return &cpus[i];
  }
  panic("unknown apicid\n");
}
```
Figure 25: proc.h cpu

```c
int
lapicid(void)
{
  if (!lapic)
    return 0;
  return lapic[ID] >> 24;
}
```
Figure 26: lapic.c lapicid

```c
void
popcli(void)
{
  if(readeflags()&FL_IF)
    panic("popcli - interruptible");
  if(--mycpu()->ncli < 0)
    panic("popcli");
  if(mycpu()->ncli == 0 && mycpu()->intena)
    sti();
}
```
Figure 27: spinlock.c popcli

```c
static inline void
sti(void)
{
  asm volatile("sti");
}
```

Figure 28: x86.h sti

When calling myproc, pushcli disables interrupts to allow mycpu to retrieve the local APIC, which in turn retrieves the desired process. popcli ends this set of actions by removing the paired pushcli.

Returning to sys_sbrk (Figure 18), the next performed function is growproc.

```c
// 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;
}
```

Figure 29: proc.c growproc

As visible, it will either allocate or deallocate n bytes from the process through the two sub-functions.

```c
// 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;
}
```

Figure 30: vm.c allocuvm

```c
#define PGROUNDUP(sz)  (((sz)+PGSIZE-1) & ~(PGSIZE-1))
```

Figure 31: mmu.h PGROUNDUP

Simple rounds up the page table given an argument.

```c
// Allocate one 4096-byte page of physical memory.
// Returns a pointer that the kernel can use.
// Returns 0 if the memory cannot be allocated.
char*
kalloc(void)
{
  struct run *r;

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  if(kmem.use_lock)
    release(&kmem.lock);
  return (char*)r;
}
```

Figure 32: kalloc.c kalloc

```c
// Acquire the lock.
// Loops (spins) until the lock is acquired.
// Holding a lock for a long time may cause
// other CPUs to waste time spinning to acquire it.
void
acquire(struct spinlock *lk)
{
  pushcli(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");

  // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0)
    ;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that the critical section's memory
  // references happen after the lock is acquired.
  __sync_synchronize();

  // Record info about lock acquisition for debugging.
  lk->cpu = mycpu();
  getcallerpcs(&lk, lk->pcs);
}
```

Figure 33: spinlock.c acquire

```c
// Check whether this cpu is holding the lock.
int
holding(struct spinlock *lock)
{
  return lock->locked && lock->cpu == mycpu();
}
```

Figure 34: spinlock.c holding

```
static inline uint
xchg(volatile uint *addr, uint newval)
{
  uint result;

  // The + in "+m" denotes a read-modify-write operand.
  asm volatile("lock; xchgl %0, %1" :
               "+m" (*addr), "=a" (result) :
               "1" (newval) :
               "cc");
  return result;
}
```

Figure 35: x86.h xchg

```
// Record the current call stack in pcs[] by following the %ebp chain.
void
getcallerpcs(void *v, uint pcs[])
{
  uint *ebp;
  int i;

  ebp = (uint*)v - 2;
  for(i = 0; i < 10; i++){
    if(ebp == 0 || ebp < (uint*)KERNBASE || ebp == (uint*)0xffffffff)
      break;
    pcs[i] = ebp[1];     // saved %eip
    ebp = (uint*)ebp[0]; // saved %ebp
  }
  for(; i < 10; i++)
    pcs[i] = 0;
}
```

Figure 36: spinlock.c getcallerpcs

```c
// Release the lock.
void
release(struct spinlock *lk)
{
  if(!holding(lk))
    panic("release");

  lk->pcs[0] = 0;
  lk->cpu = 0;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that all the stores in the critical
  // section are visible to other cores before the lock is released.
  // Both the C compiler and the hardware may re-order loads and
  // stores; __sync_synchronize() tells them both not to.
  __sync_synchronize();

  // Release the lock, equivalent to lk->locked = 0.
  // This code can't use a C assignment, since it might
  // not be atomic. A real OS would use C atomics here.
  asm volatile("movl $0, %0" : "+m" (lk->locked) : );

  popcli();
}
```
Figure 37: spinlock.c release

The process of kalloc is extensive, in which the process itself will allocate a page of memory and return the pointer. The sub-function acquire will wait for a lock (checks through holding) before continuing to manipulate the memory through moving the free pointer to next. The lock is then released and pointer is returned. In short, kalloc retrieves the free-list pointer, moves the head of said pointer to next and returns the pointer for usage.

Continuing in allocuvm (Figure 30), assuming valid memory it will allocate memory through memset:

```c
void*
memset(void *dst, int c, uint n)
{
  if ((int)dst%4 == 0 && n%4 == 0){
    c &= 0xFF;
    stosl(dst, (c<<24)|(c<<16)|(c<<8)|c, n/4);
  } else
    stosb(dst, c, n);
  return dst;
}
```

Figure 38: string.c memset

```c
static inline void
stosl(void *addr, int data, int cnt)
{
  asm volatile("cld; rep stosl" :
               "=D" (addr), "=c" (cnt) :
               "0" (addr), "1" (cnt), "a" (data) :
               "memory", "cc");
}
```

Figure 39: x86.h stosl

```c
static inline void
stosb(void *addr, int data, int cnt)
{
  asm volatile("cld; rep stosb" :
               "=D" (addr), "=c" (cnt) :
               "0" (addr), "1" (cnt), "a" (data) :
               "memory", "cc");
}
```

Figure 40: x86.h stosb

memset validates the destination against size, in which depending on condition will store the appropriate string (long vs byte).

Returning to allocuvm (Figure 30) are conditions to check allocuvm through sub-functions mappages and V2P.

```c
// 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;
}
```
Figure 41: vm.c mappages

The function creates page table entries by walking the table by increasing the page size every iteration.

```c
// 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.
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)];
}
```

Figure 42: vm.c walkpgdir

Returns valid addresses of the page table corresponding to virtual addresses through validation of the page indices.

```c
// page directory index
#define PDX(va)         (((uint)(va) >> PDXSHIFT) & 0x3FF)
```

Figure 43: mmu.h PDX

```c
#define V2P(a) (((uint) (a)) - KERNBASE)
#define P2V(a) (((void *) (a)) + KERNBASE)
```

Figure 44: memlayout.h V2P and P2V

```c
// Address in page table or page directory entry
#define PTE_ADDR(pte)   ((uint)(pte) & ~0xFFF)
```

Figure 45: mmu.h PTE_ADDR

The three functions simply return the page index (PDX given virtual address), or the desired address given the opposite. The address itself is returned through PTE_ADDR.

Returning to allocuvm (Figure 30), in the event that the condition is true or growproc was given a negative argument, the function proceeds to deallocuvm.

```c
// Deallocate user pages to bring the process size from oldsz to
// newsz.  oldsz and newsz need not be page-aligned, nor does newsz
// need to be less than oldsz.  oldsz can be larger than the actual
// process size.  Returns the new process size.
int
deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
  pte_t *pte;
  uint a, pa;

  if(newsz >= oldsz)
    return oldsz;

  a = PGROUNDUP(newsz);
  for(; a  < oldsz; a += PGSIZE){
    pte = walkpgdir(pgdir, (char*)a, 0);
    if(!pte)
      a = PGADDR(PDX(a) + 1, 0, 0) - PGSIZE;
    else if((*pte & PTE_P) != 0){
      pa = PTE_ADDR(*pte);
      if(pa == 0)
        panic("kfree");
      char *v = P2V(pa);
      kfree(v);
      *pte = 0;
    }
  }
  return newsz;
}
```
Figure 46: vm.c deallocuvm

Similar to allocuvm with walking, with the exception of kfree instead of increasing size.

```c
// Free the page of physical memory pointed at by v,
// which normally should have been returned by a
// call to kalloc().  (The exception is when
// initializing the allocator; see kinit above.)
void
kfree(char *v)
{
  struct run *r;

  if((uint)v % PGSIZE || v < end || V2P(v) >= PHYSTOP)
    panic("kfree");

  // Fill with junk to catch dangling refs.
  memset(v, 1, PGSIZE);

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = (struct run*)v;
  r->next = kmem.freelist;
  kmem.freelist = r;
  if(kmem.use_lock)
    release(&kmem.lock);
}
```
Figure 47: kalloc.c kfree

The opposite of kalloc, freeing the memory and changing the freelist accordingly.

Assuming that de/allocuvm was successful, it will return the new size to growproc (Figure 29), which will set the new size and execute switchuvm.

```c
// Switch TSS and h/w page table to correspond to process p.
void
switchuvm(struct proc *p)
{
  if(p == 0)
    panic("switchuvm: no process");
  if(p->kstack == 0)
    panic("switchuvm: no kstack");
  if(p->pgdir == 0)
    panic("switchuvm: no pgdir");

  pushcli();
  mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A, &mycpu()->ts,
                                sizeof(mycpu()->ts)-1, 0);
  mycpu()->gdt[SEG_TSS].s = 0;
  mycpu()->ts.ss0 = SEG_KDATA << 3;
  mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
  // setting IOPL=0 in eflags *and* iomb beyond the tss segment limit
  // forbids I/O instructions (e.g., inb and outb) from user space
  mycpu()->ts.iomb = (ushort) 0xFFFF;
  ltr(SEG_TSS << 3);
  lcr3(V2P(p->pgdir));  // switch to process's address space
  popcli();
}
```
Figure 48: vm.c switchuvm

```c
static inline void
ltr(ushort sel)
{
  asm volatile("ltr %0" : : "r" (sel));
}
```
Figure 49: x86.h ltr

```c
static inline void
lcr3(uint val)
{
  asm volatile("movl %0,%%cr3" : : "r" (val));
}
```
Figure 50: x86.h lcr3

Per documentation, switches the tables so to correspond to correspond to the process, done by manipulating mycpu structure.

Afterwards, growproc returns and begins recursively returning back to user. The returned integer (0 success, 1 fail) is passed as the eax variable and returned to trap.

```
  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

Figure 51: trapasm.S trapret

Everything from the stack prior to the trap call is popped, the privilege/level is set back to user, and the instruction returns back to main. The process results in either a changed process size or a returned error which can be handled accordingly. This returns up to morecore (Figure 3) and validates this pointer and returns 0 or the header freep accordingly.