Lab 2: Memory Allocation for XV6 - Alejandro Vargas (avarg116)

Task No.1: Heap Allocator

```
void
fileinit(void)
{
  initlock(&ftable.lock, "ftable");
  //ftable.file= (struct file *)bd_malloc(sizeof(struct file) * NFILE); //change
}

// Allocate a file structure.
struct file*
filealloc(void)
{
  struct file *f;

  acquire(&ftable.lock);
  f = (struct file *)bd_malloc(sizeof(struct file)); //allocate dynamic-sized memory
  release(&ftable.lock);

  if (f) {
    f->ref = 1;
  }

  return f;
```

In the initial phase, the main objective was to implement dynamic allocation through the buddy allocator. Improving xv6's memory allocation, the declaration of file[NFILE] had to be removed. I used a pointer to dynamically allocate file structures. The buddy allocator is used to allocate memory for these file structures, allowing dynamic allocation based on available memory. Within struct file* filealloc(void), I used the code inside the function as reference and modified it. Since the code was changed to use dynamic memory allocation for creating a new file structure, the loop was no longer needed.

```
void
fileclose(struct file *f)
{
  acquire(&ftable.lock);
  if (f->ref < 1)
    panic("fileclose");
  if (--f->ref > 0) {
    release(&ftable.lock);
    return;
  }
  bd_free(f); // Deallocate memory using buddy allocator
  release(&ftable.lock);
```

Using bd_malloc for allocating memory, it creates a new instance of a struct file dynamically in the heap memory. The conditional statement checks if the memory allocation was completed. ref is set to 1 or null depending on whether a given struct file object is available for use. Then I ran 'alloctest' which passed all tests. It addresses the limitations of the original xv6 design, which had a static allocation of file structures. I also simplified fileclose because ff is not needed but later learned this was not necessary.

Output:

```
init: starting sh
$ alloctest
filetest: start
filetest: OK
$ QEMU: Terminated
[avarg116@sledge cs179f-fall23]$
```

Task No.2: Lazy Page Allocation

In memory management, lazy allocation is a technique that delays the allocation of a resource until processes need it. It's advantageous because it reduces memory resources and lets processes create large sparse data structures. In xv6, the sbrk() system call allocates memory and maps it. Implementing lazy allocation for user-space heap memory, we reserve virtual memory and don't allocate immediately. When a page fault happens, it allocates a page and maps it to the faulting address.

```
uint64
sys_sbrk(void)
{

  int n;
  if(argint(0, &n) < 0)
    return -1;
     uint64 oldz= myproc()->sz; //old process size
  myproc()->sz +=n; //increase process's size
  if(n < 0) {
    uvmdealloc(myproc()->pagetable, oldz, myproc()->sz); //deallocate memory
  }

  return oldz;
}
```

sysproc.c: in sysproc.c, we were tasked with the removal of page allocation from the sbrk(n) system call. I wrote the line lineuint64 oldz = myproc()->sz; which stores the current value of the process's memory size. Similarly, we were instructed to erase the call to growproc().

```
void uvmunmap(pagetable_t pagetable, uint64 va, uint64 size, int do_free)
{
  uint64 a, last;
  pte_t *pte;
  uint64 pa;

  a = PGROUNDDOWN(va);
  last = PGROUNDDOWN(va + size - 1);
  for(;;){
    if((pte = walk(pagetable, a, 0)) == 0) {
      //panic("uvmunmap: walk");
      if(a == last)
        break;
      a += PGSIZE;
      continue;
    }
```

vm.c: I made changes to uvmunmap() so that it won't panic if all pages aren't mapped. In general, I commented out panics and wrote if(a == last) break; a += PGSIZE; continue; to avoid kernel crashes.

Additional changes included handling negative sbrk() arguments. This was accomplished through editing the line deallocating memory in the virtual memory system for the current process and return.

```
} else if (r_scause() == 13 || r_scause() == 15) { //check page fault
  uint64 faulting_address = r_stval();
  //kill process
  if(faulting_address > myproc()->sz) {
    myproc()->killed= 1;
  }
   if (faulting_address < myproc()->tf->sp) {
    myproc()->killed = 1;
   }

  uint64 page_boundary = PGROUNDDOWN(faulting_address); //calculate page boundary

  char *mem = kalloc(); //allocate new page
  if (mem == 0) {
    p->killed= 1;
  } else {
    memset(mem, 0, PGSIZE);
  //map new page to page table
  if (mappages(p->pagetable, page_boundary, PGSIZE, (uint64)mem, PTE_W | PTE_X | PTE_R | PTE_U) != 0) {
            kfree(mem);
            p->killed = 1;
        }
  }
  //edit
```

trap.c: I started by including else if (r_scause() == 13 || r_scause() == 15) to check whether a fault is a page fault in usertrap. faulting_address = r_stval(); is extracting faulting virtual addresses using r_stval(). Using kalloc(), the code proceeds to allocate a new page of physical memory. The instructions suggested stealing code from uvmalloc() in vm.c. mappages deallocates the memory if mappages fails. Next, I used PGROUNDDOWN to calculate and store the page boundary for a given faulting address.

Usertests: The goal was to make these modifications in the kernel code so that it passes both lazytests and usertests. I was not able to complete usertests in time, only passing some tests. I got stuck on pgbug and tried to fix it in copyout, copyin, and copyinstr, but it results in panic: kerneltrap.

Testing

```
$ lazytests
lazytests starting
running test lazy alloc
test lazy alloc: OK
running test lazy unmap
test lazy unmap: OK
running test out of memory
test out of memory: OK
ALL TESTS PASSED
$ usertests
usertests starting
test reparent2: OK
test pgbug: scause 0x000000000000000d
sepc=0x0000000080001592 stval=0xeaeb0b5b87f57f5e
panic: kerneltrap
```

```
Task1 grading: alloctest (50/100):
$ make qemu-gdb
OK (5.9s)
Task2 grading: lazytests (25/100):
$ make qemu-gdb
(9.5s)
  lazy: map: OK
  lazy: unmap: OK
Task2 grading: usertests (25/100):
$ make qemu-gdb
Timeout! (150.1s)
  usertests: pgbug: FAIL
    Failed pgbug
  usertests: sbrkbugs: FAIL
    Failed sbrkbugs
  usertests: argptest: FAIL
    Failed argptest
  usertests: sbrkmuch: FAIL
    Failed sbrkmuch
  usertests: sbrkfail: FAIL
    Failed sbrkfail
  usertests: sbrkarg: FAIL
    Failed sbrkarg
  usertests: stacktest: FAIL
    Failed stacktest
  usertests: all tests: FAIL
    ...
         hart 2 starting
         init: starting sh
         $ usertests
         usertests starting
         test reparent2: qemu-system-riscv64: terminating on signal 15 from pid 3705 (make)
    MISSING '^ALL TESTS PASSED$'
Score: 75/100
make: *** [grade] Error 1
[avarg116@sledge cs179f-fall23]$
```