Lab 4: Memory Mangement

4/5/2022

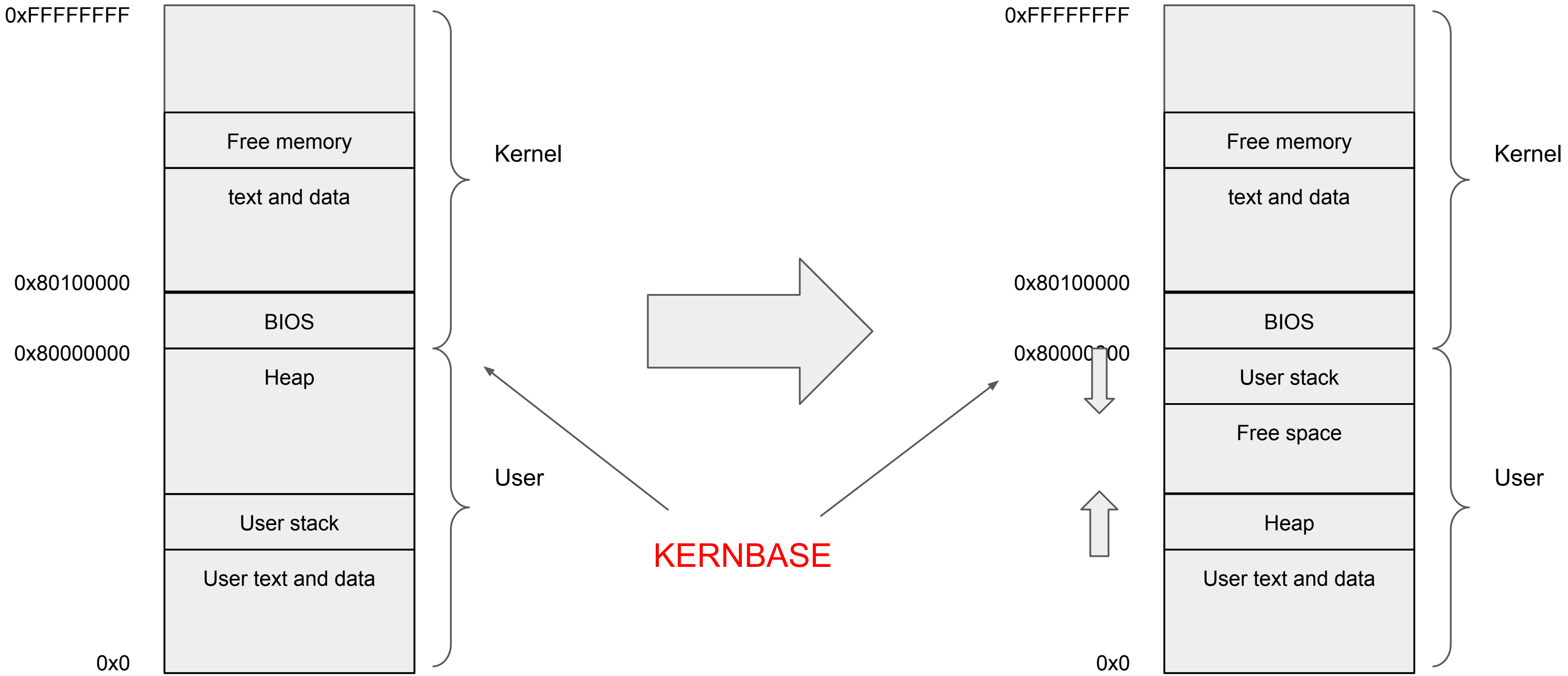
Preliminary

1. Read lab description carefully https://yueduan.github.io/cs450_lab4.html
2. Get starter code from this repo on github;

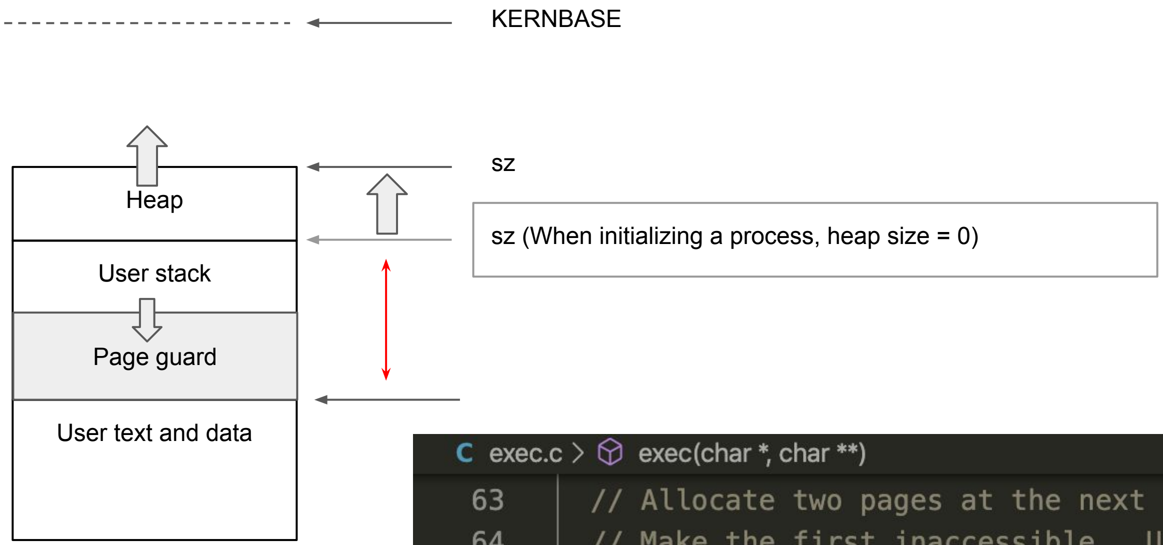
Introduction

1. Change the user memory layout. (70%)
    a. code-stack-heap -> code-heap-stack
    b. The size of stack is one page, so there will be a gap between heap and stack
2. Implement stack growth (30%)
    a. Default: one page
    b. Raise a page fault when stack grows beyond its allocated page.
    c. Currently, kernel will panic if stack overflows.
    d. In case of T_PGFLT trap, allocate new page(s) instead of panic.

Memory layout



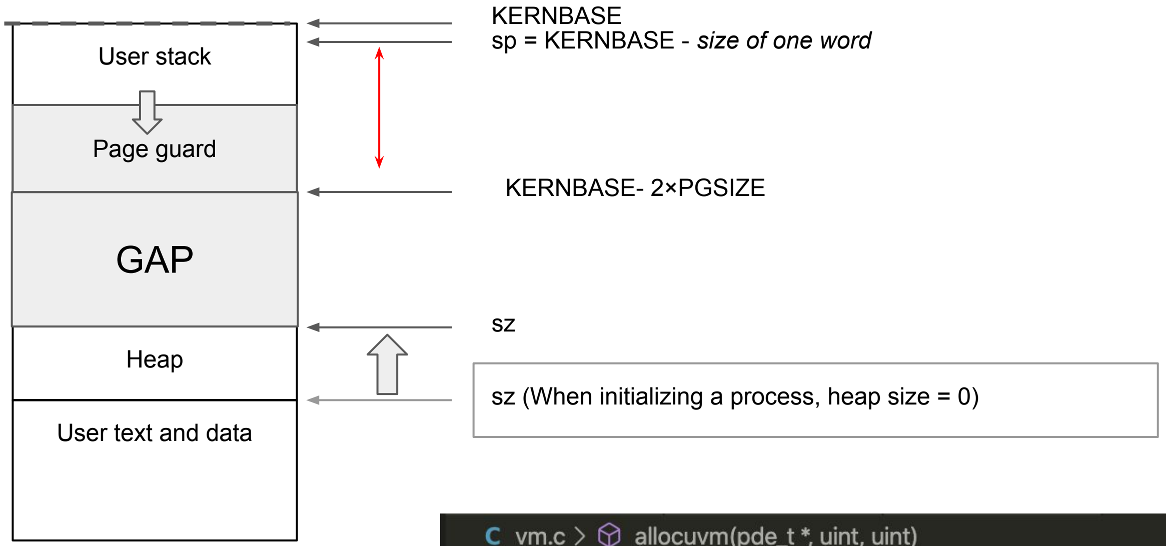
Memory Layout (before)



exec.c > exec(char *, char **)

```c
// Allocate two pages at the next page boundary.
// Make the first inaccessible.  Use the second as the user stack.
sz = PGROUNDUP(sz);
if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
  goto bad;
clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
sp = sz;
```

Memory Layout (after)



C vm.c > allocuvm(pde_t *, uint, uint)

```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)
```

Implementation

1. Modify exec() function in exec.c (line 63) (please read allocuvm() in vm.c)
   a. sz and sp
2. Modify arg functions in syscall.c (fetchint(), fetchstr() and argptr())
   a. curproc->sz
3. Modify copyuvm() in vm.c
   a. sz
4. Write a simple test of your relocated stack (test argc in main() function)
   a. printf("%p", &argc)
5. Add case in trap.c to handle T_PGFLT by allocating new page(s) to the stack
   a. rcr2() reads the address that caused the page fault from register CR2, which should be from the page directly below the current bottom of the stack
6. Test stack growth using the recursive test code linked to on the last slide

How does exec() work with memory currently?

We can break it down into six parts:

1. Opens the exexutable and parses it
2. Initializes kernel memory using `setupkvm()`
3. Loads the program into memory using `allocuvm()` and `loaduvm()`
4. Allocates space in memory for the user stack using `allocuvm()` again
5. Initialize the stack pointer and push arguments to the stack
6. Set up the process struct with its new page table, size and trap frame, switch to the new page table and free the old one (originally copied from the parent)

How does exec() work with memory currently?

We can break it down into six parts:

1. Opens the exexutable and parses it

```
int
exec(char *path, char **argv)
{
  char *s, *last;
  int i, off;
  uint argc, sz, sp, ustack[3+MAXARG+1];
  struct elfhdr elf;
  struct inode *ip;
  struct proghdr ph;
  pde_t *pgdir, *oldpgdir;
  struct proc *curproc = myproc();

  begin_op();

  if((ip = namei(path)) == 0){
    end_op();
    cprintf("exec: fail\n");
    return -1;
  }
  ilock(ip);
  pgdir = 0;

  // Check ELF header
  if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
    goto bad;
  if(elf.magic != ELF_MAGIC)
    goto bad;
  ...
```

NO CHANGES

How does exec() work with memory currently?

We can break it down into six parts:

2. Initializes kernel memory using `setupkvm()`

```c
int
exec(char *path, char **argv)
{
  ...
  if((pgdir = setupkvm()) == 0)
    goto bad;
```

```c
// vm.c
// Set up kernel part of a page table.
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;
  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
  if (P2V(PHYSTOP) > (void*)DEVSPACE)
    panic("PHYSTOP too high");
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0) {
      freevm(pgdir);
      return 0;
    }
  return pgdir;
}
```

NO CHANGES

How does exec() work with memory currently?

We can break it down into six parts:

3. Loads the program into memory using `allocuvm()` and `loaduvm()`

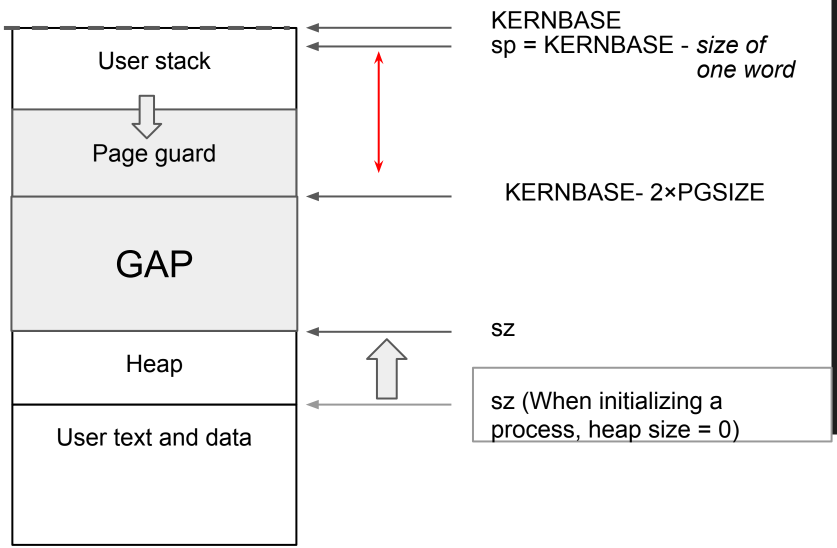
```
int
exec(char *path, char **argv)
{
  ...
  // Load program into memory.
  sz = 0;
  for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
      goto bad;
    if(ph.type != ELF_PROG_LOAD)
      continue;
    if(ph.memsz < ph.filesz)
      goto bad;
    if(ph.vaddr + ph.memsz < ph.vaddr)
      goto bad;
    if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
      goto bad;
    if(ph.vaddr % PGSIZE != 0)
      goto bad;
    if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
      goto bad;
  }
  iunlockput(ip);
  end_op();
  ip = 0;
  ...
```

NO CHANGES

How does exec() work with memory currently?

We can break it down into six parts:

4. Allocates space in memory for the user stack using `allocuvm()` again




```
int
exec(char *path, char **argv)
{
  ...
 // Allocate two pages at the next page boundary.
 // Make the first inaccessible.  Use the second as the user stack.
 sz = PGROUNDUP(sz);
 if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
   goto bad;
 clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
 sp = sz;
 ...
```

CHANGES!

How does exec() work with memory currently?

We can break it down into six parts:

5. Initialize the stack pointer and push arguments to the stack

```
int
exec(char *path, char **argv)
{
  ...
  sp = sz;    Mentioned in previous slide. Otherwise...

  // Push argument strings, prepare rest of stack in ustack.
  for(argc = 0; argv[argc]; argc++) {
    if(argc >= MAXARG)
      goto bad;
    sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
    if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
      goto bad;
    ustack[3+argc] = sp;
  }
  ustack[3+argc] = 0;

  ustack[0] = 0xffffffff;  // fake return PC
  ustack[1] = argc;
  ustack[2] = sp - (argc+1)*4;  // argv pointer

  sp -= (3+argc+1) * 4;
  if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
    goto bad;
  ...
```

NO CHANGES

How does exec() work with memory currently?

We can break it down into six parts:

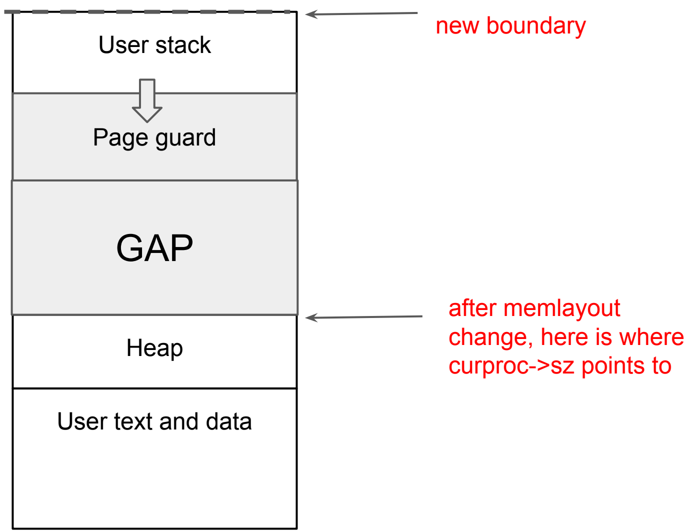
6. Set up the process struct with its new page table, size and trap frame, switch to the new page table and free the old one (originally copied from the parent)

```
int
exec(char *path, char **argv)
{ ...
  // Save program name for debugging.
  for(last=s=path; *s; s++)
    if(*s == '/')
      last = s+1;
  safestrcpy(curproc->name, last, sizeof(curproc->name));

  // Commit to the user image.
  oldpgdir = curproc->pgdir;
  curproc->pgdir = pgdir;
  curproc->sz = sz;
  curproc->tf->eip = elf.entry;  // main
  curproc->tf->esp = sp;
  switchuvm(curproc);
  freevm(oldpgdir);
  return 0;

bad:
  if(pgdir)
    freevm(pgdir);
  if(ip){
    iunlockput(ip);
    end_op();
  }
  return -1;
}
```

Don’t change these, but we’ll need to add a new property to `proc struct` to track the number of user stack pages

Hint for changes in fetchint/fetchstr/argptr

- Xv6 tracks the size of a proc's address space using curproc->sz
- In fetchint/fetchstr/argptr where an argument passed in is within legal address range
    - you now need another field to track the stack;




```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;
}
```

Hint for changes in copyuvm

- Copyuvm() function is used to make a copy of parent proc's virtual memory for the child proc;
- Previously the virtual memory is consecutive from 0 to curproc()->sz;
- Now:
    - memory from 0 to curproc()->sz contains code and heap;
    - one-page stack grow from KERNBASE towards 0 followed by a page guard;
- You need extra handling to copy the stack page (& page guard);

Hint for stack growth

- Trying to access the inaccessible page between the user data section and the current user stack causes a page fault, or `T_PGFLT`.
- At the moment our trap handler does not do anything in the case of a page fault, so it falls into the default case.
- To implement stack growth, you will need to add a case for `T_PGFLT`, check what address caused it, and allocate a new page only if the bad address is from the page right below the stack.

```
void
trap(struct trapframe *tf)
{
 if(tf->trapno == T_SYSCALL){
   ...
 }

 switch(tf->trapno){
 case T_IRQ0 + IRQ_TIMER:
   ...
   break;
 case T_IRQ0 + IRQ_IDE:
   ...
   break;
 case T_IRQ0 + IRQ_IDE+1:
   ...
   break;
 case T_IRQ0 + IRQ_KBD:
   ...
   break;
 case T_IRQ0 + IRQ_COM1:
   ...
   break;
 case T_IRQ0 + 7:
 case T_IRQ0 + IRQ_SPURIOUS:
   ...
   break;

 default:
   ...
 }
```

Test prog example

```c
#include "types.h"
#include "stat.h"
#include "user.h"

int main (int argc, char* argv[]){
    int v = argc;
    printf(1, "%p\n", &v);
    exit(0);
}
```

```
Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start 58
init: starting sh
$ lab3
7FFFFFDC
$ lab3 2 3
7FFFFFCC
$ lab3 150 250 450
7FFFFFBC
$ lab3 1 2 3 4 5 6 7 8
7FFFFF9C
$
```

Test prog for stack overflow (& bonus)

- Test file:
  https://drive.google.com/file/d/12uZUddvV9IMAKHI7HKs02EqJEImACP9W/view?usp=sharing

```
$ lab3
Usage: lab3 levels
$ lab3 100
Lab 3: Recursing 100 levels
Lab3: Yielded a value of 5050
$ lab3 1000
Lab 3: Recursing 1000 levels
Increased stack size
Increased stack size
Increased stack size
Increased stack size
Increased stack size
Increased stack size
Increased stack size
Lab3: Yielded a value of 500500
$
```

```c
#include "types.h"
#include "user.h"

// Prevent this function from being optimized, which might give it closed form
#pragma GCC push options
#pragma GCC optimize ("O0")
static int
recurse(int n)
{
 if(n == 0)
   return 0;
 return n + recurse(n - 1);
}
#pragma GCC pop_options

int
main(int argc, char *argv[])
{
 int n, m;

 if(argc != 2){
   printf(1, "Usage: %s levels\n", argv[0]);
   exit();
 }

 n = atoi(argv[1]);
 printf(1, "Lab 3: Recursing %d levels\n", n);
 m = recurse(n);
 printf(1, "Lab 3: Yielded a value of %d\n", m);
 exit();
}
```