实验 5:修改 xv6 内存布局

郑昌乾　1120223563 33.33%

杜恩俊：1120223455 33.33%

严小桐　1120223563 33.33%

目录

一、实验目标

1. 修改 xv6 的内存布局，将栈移至地址空间的顶部，并实现栈从顶部向下增长。这部分要求重新设计用户内存布局，使其更接近 Linux 的结构。
2. 实现栈的自动增长机制。当栈增长超出当前分配的页面时，应触发缺页中断并分配新的页面(help-lab5)(Lab 5 Memory Management)。
3. 实现栈增长到堆的能力。如果无法实现，需详细解释原因并展示相关代码(Lab 5 Memory Management)。

二、配置环境

1. 实验环境

(1) 虚拟化平台:Unraid 6.12.13 远程运行

(2) 操作系统:Ubuntu 22.04.4 LTS (GNU/Linux 6.8.0-40-generic x86_64)

(3) 小型操作系统内核:xv6

2. 配置 gcc 环境

(1) 输入 `sudo apt update` 更新软件包列表

(2) 输入 `sudo apt-get install -y build-essential git gcc-multilib` 配置 gcc

(3) 输入 `objdump -i` 检查是否支持 64 位:、

```
user@Lab:~$ objdump -i
BFD header file version (GNU Binutils for Ubuntu) 2.38
elf64-x86-64
```

(4) 输入 `gcc -m32 -print-libgcc-file-name` 查看 32 位 gcc 库文件路径:

```
user:~$ gcc -m32 -print-libgcc-file-name
/usr/lib/gcc/x86_64-linux-gnu/11/32/libgcc.a
```

出现上述输出，gcc 环境已配置好

3. 配置 qemu 以及 xv6 环境

(1) 输入 `sudo apt-get install qemu-system` 安装 qemu

(2) 输入 `qemu-system-i386 --version` 查看 qemu 版本:

```
user@Lab:~$ qemu-system-i386 --version
QEMU emulator version 6.2.0 (Debian 1:6.2+dfsg-2ubuntu6.22)
Copyright (c) 2003-2021 Fabrice Bellard and the QEMU Project developers
```

出现上述输出，qemu 环境已配置好

（3）输入 `git clone https://github.com/mit-pdos/xv6-public.git` 下载 xv6 系统

```
+0 records in
+0 records out
512 bytes (512 B) copied, 0.000543844 s, 938 kB/s
dd if=kernal of=xf6.img seek=1 conv=notrunc
393+1 records in
393+1 records out
```

文件最后出现输出，说明操作系统镜像文件已准备好：

4. 启动 qemu

（1）输入 `~/xv6$ echo "add-auto-load-safe-path $HOME/xv6/.gdbinit" > ~/.gdbinit` 配置 gdb

（2）输入 `make qemu` 启动 qemu，出现如下报错：

```
qemu-system-i386 -serial mon:stdio -drive file=fs.img, index=1, media=disk, format=raw -drive
file=xv6.img, index=0, media=disk, format=raw -smp 2 -m 512
gtk initialization failed
make: *** [Makefile:225: qemu] Error 1
```

（3）经查询，其原因通常与 QEMU 的 GTK 版本有关，这可能是由于缺少 GTK 库或没有正确配置显示环境引起的。在此处我们选择采用 VNC 后端启动 QEMU，通过修改 `Makefile` 其中的 `QEMUOPTS` 变量如下，修改启动方式

```
QEMUOPTS = -drive file=fs.img, index=1, media=disk, format=raw -drive
file=xv6.img, index=0, media=disk, format=raw -smp $(CPUS) -m 512 -nographic
```

（4）命令行中输入 `make qemu` 启动 qemu，进入 qemu 界面：

```
SeaBIOS (version 1.15.0-1)



iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+1FF8B4A0+1FECB4A0 CA00
```

```
    Booting from Hard Disk..xv6...
    cpu0: starting 0
    sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap
sta8
    init: starting sh
    $
```

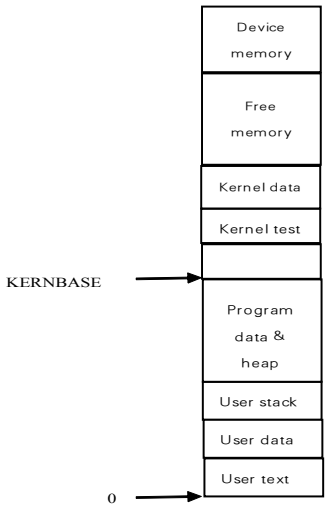
三、xv6 功能分析

1. 内存布局:

在 xv6 中，内存布局分为内核空间和用户空间，

内核位于内存空间的高地址部分，从一个固定点（`KERNBASE`，通常是 `0x80000000`）开始。包含内核代码和数据、I/O 内存映射、内核栈、物理内存映射

用户空间位于地址空间的低地址部分（低于 `KERNBASE`），主要包括文本段、数据段、堆段和栈段。其中栈段由中间的某个地址开始向上增长，本次实验的目的就是讲栈的分配方式由从下往上修改为从上往下。具体的内存分布见下图:



2. 内存管理相关的文件

(1) `vm.c`:

负责虚拟内存管理，包括设置内核页表、分配和释放用户内存、复制进程内存等。

(2) `proc.c`

管理进程状态，包括创建新进程、扩展内存、执行新程序、复制父进程的内存布局等

(3) `trap.c`：
处理异常和中断，包括页面错误（Page Fault）。

(4) `exec.c`
加载可执行文件和初始化进程的内存布局

(5) `syscall.c`
处理系统调用，调用相应的内存管理功能，负责在用户空间和内核空间之间切换。

(6) `memlayout.h`
定义内存布局的常量，包括内核空间和用户空间的起始和结束地址

四、代码修改

1. 具体步骤

(1) 调整用户地址空间布局。

(2) 修改 `exec.c` 以在高地址处分配栈。

(3) 调整 `proc` 结构，以正确跟踪用户栈和堆的边界。

(4) 在 `vm.c` 修改与内存管理相关的函数，确保它们能正确处理栈和堆的分配及地址验证。

(5) 修改 `trap.c` 以及 `syscall.c` 等文件实现栈增长的处理

2. 调整用户地址布局

用户地址布局的核心文件是 `memlayout.h`，它定义了内核和用户内存空间的布局。在 `memlayout.h` 中，`KERNBASE` 定义了用户地址空间的上限，在此处添加用户页顶端地址：

```
    // Key addresses for address space layout (see kmap in vm.c for layout)
    #define KERNBASE 0x80000000         // First kernel virtual address
    #define USERTOP 0x7FFFF000 // User address space top,  leaving room for stack
growth
```

3. 修改 exec.c 以在高地址处分配栈。

在 `exec.c` 中，找到分配用户栈的代码，将栈分配到代码和堆的末尾，并分配两页 `(2*PGSIZE)` 大小的栈空间；修改栈的位置到 `KERNBASE - *PGSIZE` 开始

```
    // Allocate user stack at the top of the address space.
      uint stackbase = USERTOP - *PGSIZE;
      if((allocuvm(pgdir, stackbase, USERTOP)) == 0)
        goto bad;
```

```
  clearpteu(pgdir, (char*)(stackbase));
  sp = USERTOP;
```

4. 调整 proc 结构，以正确跟踪用户栈和堆的边界。

当前 `proc->sz` 跟踪整个用户地址空间的大小，包括代码段、堆和栈。在修改后，`proc->sz` 只用于跟踪代码段和堆的大小，需要额外的字段来跟踪栈的起始位置。 故在 `proc.h` 中对 `proc` 结构设置新的结构项：

```
// Per-process state
struct proc {
  uint sz;                  // Size of process memory (bytes)
  pde_t* pgdir;              // Page table
  char *kstack;              // Bottom of kernel stack for this process
  enum procstate state;       // Process state
  int pid;                  // Process ID
  struct proc *parent;        // Parent process
  struct trapframe *tf;       // Trap frame for current syscall
  struct context *context;    // swtch() here to run process
  void *chan;               // If non-zero, sleeping on chan
  int killed;               // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;          // Current directory
  char name[16];             // Process name (debugging)
  uint stackbase;            // Base of the stack
};
```

5. 在 vm.c 等文件修改与内存管理相关的函数，确保它们能正确处理栈和堆的分配及地址验证。

在 `vm.c` 中修改 `copyuvm` 函数,以确保在 `fork` 时正确地拷贝栈：

```
copyuvm(pde_t *pgdir, uint sz, uint stackbase)
{
  ...
  // Copy the stack
  for(i = USERTOP - PGSIZE; i > stackbase  i -= PGSIZE){
   if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
     panic("copyuvm: pte should exist");
```

```
      if(!(*pte & PTE_P))
        panic("copyuvm: page not present");
      pa = PTE_ADDR(*pte);
      flags = PTE_FLAGS(*pte);
      if((mem = kalloc()) == 0)
        goto bad;
      memmove(mem, (char*)P2V(pa), PGSIZE);
      if(mappages(d, (void*)i, PGSIZE, V2P(mem), flags) < 0) {
        kfree(mem);
        goto bad;
      }
    }

    return d;
    ...
  }
```

在 `proc.c` 中，修改 `fork` 函数以传递 `stackbase` 参数给 `copyuvm` 函数

```
if((np->pgdir = copyuvm(curproc->pgdir, curproc->sz, curproc->stackbase)) == 0)
```

6. 修改 `trap.c` 以及 `syscall.c` 等文件实现栈增长的处理

在 `trap.c` 中,中处理页面错误，检测是否是栈溢出，并分配新的栈页，并在 `syscall.c` 中修改所有引用 `sz` 的地方，以反映新的栈位置。

```
  // 新增触发页错误（Page Fault）的情况
  case T_PGFLT:

    if (rcr2() < USERTOP)// 条件检查:确保页面错误地址在用户栈范围内。
    {
      cprintf("page error %x ",rcr2());
      cprintf("stack pos : %x\n", myproc()->stackbase);
      // 为用户栈分配新的页面
      if ((myproc()->stackbase = allocuvm(myproc()->pgdir,
myproc()->stackbase - 1 * PGSIZE,
      myproc()->stackbase)) == 0)
      {
```

```
      myproc()->killed = 1;
    }
    myproc()->stackbase-=PGSIZE; // 更新用户栈的栈顶位置
    cprintf("create a new page %x\n", myproc()->stackbase);
     //clearpteu(myproc()->pgdir, (char *) (myproc()->stackbase - PGSIZE));
    return;
  }
  else
  {
     myproc()->killed = 1;
    break;
  }
```

五、实验测试

1. 编写测试程序 testcase.c

(1) 主要结构:

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

// 获取当前栈指针
uint get_stack_pointer() {
    ...
}

// 深度递归调用以测试更大规模的栈增长
void recursion(int depth) {
    ...
}

// 测试深度递归引发栈增长
```

```
void test_stack_growth() {
    ...
}

// 测试堆与栈的冲突
void test_stack_heap_collision() {
    ...
}

int main(int argc, char *argv[]) {
    test_stack_growth();
    test_stack_heap_collision();
    exit();
}
```

(2) 测试栈的增长以及缺页分配

编写如下程序： 其中，`get_stack_pointer`用于获取栈指针，`recursion`函数用于递归调用，`test_stack_growth`函数用于测试栈的增长。对于每次递归调用，输出当前栈指针。

```
// 获取当前栈指针
uint get_stack_pointer() {
    uint sp;
    asm volatile("movl %%esp, %0" : "=r" (sp));
    return sp;
}

// 深度递归调用以测试更大规模的栈增长
void recursion(int depth) {
    char buffer[4096];  // 分配 4KB 的空间，确保每次递归占用整页
    memset(buffer, 1 , 4096); // 进行操作避免被优化掉
    uint sp = get_stack_pointer();
    printf(1, "Recursion depth: %d, stack address: 0x%x\n", depth , sp );

    // 在深度达到一定值之前递归
    if (depth < 100) {
```

```
        recursion(depth + 1);
    }
}

// 测试深度递归引发栈增长
void test_stack_growth() {
    printf(1, "\n=== Test: Stack Growth ===\n");
    printf(1, "Starting recursion test...\n");

    recursion(1);  // 初始递归调用，测试栈增长

    printf(1, "Stack recursion test completed.\n");
}
```

(3) 测试堆与栈的冲突

在测试栈增长的基础上，添加`test_stack_heap_collision`函数，通过分配大量堆内存，测试堆与栈的冲突。

```
// 测试堆与栈的冲突
void test_stack_heap_collision() {
    printf(1, "\n=== Test: Stack-Heap Collision ===\n");
    printf(1, "Starting stack-heap collision test...\n");
    printf(1, "Allocating 222MB of memory on the heap...\n");
    int * p = (int *)malloc(222*1024*1024 - 512*1024);
    *p = 1;

    printf(1, "Making recursions .\n");
    recursion(1);  // 初始递归调用，测试栈增长
    printf(1, "Stack recursion test completed.\n");
}
```

2. 运行测试程序

编译并运行测试程序:

```
$ make qemu-gdb
```

在`Makefile`中添加编译`testcase.c`的命令:

```
UPROGS=\
  _testcase\
```

3. 实验结果

(1) 测试栈的增长

```
=== Test: Stack Growth ===
Starting recursion test...
page error 7fffdf90 stack pos : 7fffe000
create a new page 7fffd000
page error 7fffcf90 stack pos : 7fffd000
create a new page 7fffc000
page error 7fffbf90 stack pos : 7fffc000
create a new page 7fffb000
Recursion depth: 1, stack address: 0x7FFFBF90,buffer address: 0x7FFFBF90
Recursion depth: 2, stack address: 0x7FFFBF90,buffer address: 0x7FFFCF90
Recursion depth: 3, stack address: 0x7FFFBF90,buffer address: 0x7FFFDF90
page error 7fffaf70 stack pos : 7fffb000
...
page error 7ff9ab70 stack pos : 7ff9b000
create a new page 7ff9a000
page error 7ff99b70 stack pos : 7ff9a000
create a new page 7ff99000
page error 7ff98b70 stack pos : 7ff99000
create a new page 7ff98000
Recursion depth: 100, stack address: 0x7FF98B70,buffer address: 0x7FF98B70
Stack recursion test completed
```

从输出中可以看出，随着递归深度的增加，由于栈空间不足，系统会不断分配新的页，并且对于每次递归，系统都会输出栈帧地址和站上的数据，我们可以看到栈地址逐渐从高地址向低地址移动，说明我们的栈是向下增长的。

(2) 测试堆与栈的冲突

在 `allocuvm` 函数中添加报错信息，指示栈是否增加到堆区

```
=== Test: Stack-Heap Collision ===
Starting stack-heap collision test...
Allocating 222MB of memory on the heap...
Making recursions .
page error 7fffdf90 stack pos : 7fffe000
```

```
create a new page 7fffd000
page error 7fffcf90 stack pos : 7fffd000
create a new page 7fffc000
page error 7fffbf90 stack pos : 7fffc000
create a new page 7fffb000
Recursion depth: 1, stack address: 0x7FFFBF80
Recursion depth: 2, stack address: 0x7FFFBF80
Recursion depth: 3, stack address: 0x7FFFBF80
page error 7fffaf70 stack pos : 7fffb000
create a new page 7fffa000
page error 7fff9f70 stack pos : 7fffa000
create a new page 7fff9000
...
page error 7ffe2e70 stack pos : 7ffe3000
create a new page 7ffe2000
page error 7ffe1e70 stack pos : 7ffe2000
create a new page 7ffe1000
page error 7ffe0e70 stack pos : 7ffe1000
create a new page 7ffe0000
Recursion depth: 28, stack address: 0x7FFE0E60
Recursion depth: 29, stack address: 0x7FFE0E60
Recursion depth: 30, stack address: 0x7FFE0E60
page error 7ffdfe50 stack pos : 7ffe0000
allocuvm out of memory
The stack has grown into the heap space.
create a new page fffff000
stack is allocated in wrong place, End the process!
```

从输出中可以看出，当递归深度达到 28 时，此时栈已经进入堆空间。由于我们的程序在出现这种情况时选择重新分配栈空间，而在此时，由于无空间可分，最终栈被分入错误的地址，导致程序崩溃。