lazy allocation

> 必读资料：
>
> - Lec08 - 8.1 Page Fault Basics
> - Lec08 - 8.2 Lazy page allocation
> - Lec08 - 8.3 Zero Fill On Demand
> - Lecture Notes - Xiao Fan's Personal Page (fanxiao.tech)

1. Page Faults

当试图访问 `PTE_V` 为 0 的虚拟地址或 `user` 访问 `PTE_U` 为 0 或 `kernel` 访问 `PTE_U` 为 1 以及其他违反 `PTE_W/PTE_R` 等 `flag` 的情况下会出现 `page faults`。`Page faults` 是一个 `exception`，总共有 3 种 `page faults`：

- `load page faults`：当 `load instruction` 无法翻译虚拟地址时发生
- `store page faults`：当 `store instruction` 无法翻译虚拟地址时发生
- `instruction page faults`：当一个 `instruction` 的地址无法翻译时发生

`page faults` 种类的代码存放在 `scause` 寄存器中，无法翻译的地址存放在 `stval` 寄存器中。

在 `xv6` 中对于 `exception` 一律都会将这个进程 `kill` 掉，但是实际上可以结合 `page faults` 实现一些功能。

1. 可以实现 `copy-on-write fork`。在 `fork` 时，一般都是将父进程的所有 `user memory` 复制到子进程中，但是 `fork` 之后一般会直接进行 `exec`，这就会导致复制过来的 `user memory` 又被放弃掉。因此改进的思路是：子进程和父进程共享一个物理内存，但是 `mapping` 时将 `PTE_W` 置零，只有当子进程或者父进程的其中一个进程需要向这个地址写入时产生 `page fault`，此时才会进行 `copy`
2. 可以实现 `lazy allocation`。旧的 `sbrk()` 申请分配内存，但是申请的这些内存进程很可能不会全部用到，因此改进方案为：当进程调用 `sbrk()` 时，将修改 `p->sz`，但是并不实际分配内存，并且将 `PTE_V` 置 0。当在试图访问这些新的地址时发生 `page fault` 再进行物理内存的分配
3. `paging from disk`：当内存没有足够的物理空间时，可以先将数据存储在其他的存储介质（比如硬盘）上，，将该地址的 `PTE` 设置为 `invalid`，使其成为一个 `evicted page`。当需要读或者写这个 `PTE` 时，产生 `Page fault`，然后在内存上分配一个物理地址，将这个硬盘上的 `evicted page` 的内容写入到该内存上，设置 `PTE` 为 `valid` 并且引用到这个内存物理地址

2. Supervisor Cause Register (scause)

PDF 版传送门

Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2–3	*Reserved*
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6–7	*Reserved*
1	8	User external interrupt
1	9	Supervisor external interrupt
1	≥10	*Reserved*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	*Reserved*
0	5	Load access fault
0	6	AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call
0	9–11	*Reserved*
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	≥16	*Reserved*

Table 4.2: Supervisor cause register (`scause`) values after trap.

3. Lab

`lab1`、`lab2` 和 `lab3` 其实是一个整体

3.1 原理讲解

`xv6` 中通过调用 `sbrk()` 实现调整页面大小的分配，`sbrk(n)` 系统调用将进程的内存大小增加 n 个字节，然后返回新分配区域的开头 (即旧大小)，主要源码在 `kernel/sysproc.c` 中的 `sys_sbrk()`，`sys_sbrk()` 其实就是通过调用 `growproc()` 去增长内存大小的

`lazy allocation` 就是当进程在申请内存时，我们只是记录当前进程的申请内存的大小 (只改变 `p->sz`)，但是并不会真正调用 `growproc()` 去改变内存大小；而是当进程在使用这块内存时再去 `growproc()`

当进程在访问一个地址时，而此地址并没有真正的被分配在内存中，那么 `xv6` 就会产生两种中断

- `r_scause() 为 13`：`Load page fault`
- `r_scause() 为 15`：`Store/AMO page fault`

当在 `usertrap()` 中收到这两种中断后，可以通过 `r_stval()` 获取使得产生中断的虚拟地址

拿到虚拟地址后，我们就需要为这个虚拟地址在物理内存中开辟一个空间并进行映射

3.2 代码实现

首先修改 `sys_sbrk()` 使其只增加 `p->sz`，而不真正分配内存；这里需要注意 n 有可能为负数，如果为负数的话相当于缩小进程内存大小，那么有可能被去掉那部分内存已经真正映射到了物理内存，此时就需要释放

```
  if(argint(0, &n) < 0)
    return -1;

+ struct proc *p = myproc();
+ addr = p->sz;
+
+ if(n >= 0)
+   p->sz += n;
+ else  // n < 0
+   p->sz = uvmdealloc(p->pagetable, p->sz, p->sz + n);
+
+ // if(growproc(n) < 0)
+ //   return -1;
  return addr;
```

修改 `usertrap()` 支持 `r_scause()` 为 13 和 15 的情况，首先拿到对应的虚拟地址，这里我们单独写一个方法 `lazy_allocate` 去负责对虚拟地址 `va` 进行分配内存

```
  } else if((which_dev = devintr()) != 0){
    // ok
+ } else if(r_scause() == 13 || r_scause() == 15) {
+   uint64 va = r_stval();
+   if(lazy_allocate(va) != 0)
+     goto bad;
  } else {
+ bad:
    printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid);
```

在 `kernel/defs.h` 中声明 `lazy_allocate()`

```
void            procdump(void);
+ int              lazy_allocate(uint64 va);
```

参考 `uvmalloc` 方法，在 `kernel/proc.c` 末尾实现 `lazy_allocate()`

```
+ int
+ lazy_allocate(uint64 va)
+ {
+   struct proc *p = myproc();
+   if(va < p->trapframe->sp || va >= p->sz)
+     return -1;
+
+   va = PGROUNDDOWN(va);
+   char *mem = kalloc();
+   if(mem == 0)
```

```
+     return -1;
+
+   memset(mem, 0, PGSIZE);
+   if(mappages(p->pagetable, va, PGSIZE, (uint64)mem, PTE_W | PTE_X | PTE_R |
PTE_U) != 0)  {
+     kfree(mem);
+     uvmunmap(p->pagetable, va, 1, 1);
+     return -1;
+   }
+
+   return 0;
+ }
```

根据实验要求中的 `hint` 修改 `vm.c`

首先是 `walkaddr()`，`usertests.c` 中的 `sbrkarg()` 测试会报错 `write sbrk failed`

经过查看发现，`write` 方法最终会调用 `walkaddr` 方法导致找不到虚拟地址对应的 `PTE`，所以需要修改 `walkaddr()`

```
  if(va >= MAXVA)
    return 0;

  pte = walk(pagetable, va, 0);
+  if(pte == 0 || (*pte & PTE_V) == 0) {
+    if(lazy_allocate(va) != 0) {
+      return 0;
+    }
+    pte = walk(pagetable, va, 0);  // lazy allocate 后再次 walk
+  }

  if(pte == 0)
```

修改 `uvmunmap()`

```
    if((pte = walk(pagetable, a, 0)) == 0)
+     continue;
+     // panic("uvmunmap: walk");
    if((*pte & PTE_V) == 0)
+     continue;
+     // panic("uvmunmap: not mapped");
    if(PTE_FLAGS(*pte) == PTE_V)
```

修改 `uvmcopy`

```
    if((pte = walk(old, i, 0)) == 0)
+     continue;
+     // panic("uvmcopy: pte should exist");
    if((*pte & PTE_V) == 0)
+     continue;
+     // panic("uvmcopy: page not present");
    pa = PTE2PA(*pte);
```