Lab05: xv6 lazy page allocation

Eliminate Allocation from `sbrk()` (Easy)

1) 实验目的

本实验的目的是修改 `sbrk()` 系统调用的实现，使其不再进行内存分配，只负责更新进程的内存大小。通过这个实验，你将理解如何管理进程的虚拟内存以及可能会引发的内存错误。

2) 实验步骤

1. **修改 `sys_sbrk()` 函数**:
   - **在 `kernel/sysproc.c` 中找到 `sys_sbrk()` 实现**:

```
int sys_sbrk(void) {
  int n;
  if (argint(0, &n) < 0)
    return -1;
  if (n < 0) {
    // Shrinking memory size
    if (myproc()->sz + n < 0)
      return -1;
    if (uvmunmap(myproc()->pagetable, myproc()->sz + n, -n, 1) < 0)
      return -1;
  } else {
    // Growing memory size
    if (growproc(n) < 0)
      return -1;
  }
  int oldsz = myproc()->sz;
  myproc()->sz += n;
  return oldsz;
}
```

   - **删除内存分配相关的代码**:

```
int sys_sbrk(void) {
  int n;
  if (argint(0, &n) < 0)
    return -1;

  // 直接更新进程的内存大小，不进行内存分配
  int oldsz = myproc()->sz;
  myproc()->sz += n;

  return oldsz;
}
```

2. **重启 xv6 并测试**：

   - **编译和启动 xv6**：

     ```
     make fs.img
     qemu-system-riscv64 -drive file=fs.img,format=raw
     ```

   - **在 shell 中运行 `echo hi`**：

     ```
     init: starting sh
     $ echo hi
     ```

3. **分析错误信息**：

   - **错误信息**：

     ```
     usertrap(): unexpected scause 0x000000000000000f pid=3
                 sepc=0x0000000000001258 stval=0x0000000000004008
                 va=0x0000000000004000 pte=0x0000000000000000
     panic: uvmunmap: not mapped
     ```

   - **分析原因**： 该错误信息表明进程在执行 `echo hi` 时发生了页错误（page fault）。由于 `sbrk()` 系统调用只更新了进程的虚拟内存大小，并没有实际分配内存，因此访问虚拟地址 `0x4008` 时发生了页错误。

     当你只更新进程的内存大小而不进行实际的内存分配时，用户进程访问这些未分配的内存区域会引发页错误。具体来说，`uvmunmap` 函数试图释放一个未映射的内存区域，导致系统崩溃。

3) 实验中遇到的困难和解决办法

1. **页错误和内存分配不一致**：

   - **困难描述**：在修改 `sbrk()` 之后，程序运行时会引发页错误（page fault）。错误信息显示虚拟地址 `0x4008` 没有映射到任何物理内存区域，这表明内存区域虽然在进程的虚拟地址空间中存在，但实际上没有分配任何物理内存。
   - **解决办法**：为了避免页错误，需要确保在进程的虚拟地址空间中实际分配内存。在只更新虚拟地址空间大小而不进行实际内存分配时，用户进程尝试访问这些未分配的内存区域会引发错误。可以通过恢复 `sbrk()` 中的内存分配逻辑来解决这个问题，确保在增加虚拟内存时也进行实际的内存分配。

2. **内存访问和映射错误**：

   - **困难描述**：由于 `sbrk()` 仅更新了进程的内存大小，而没有实际进行内存映射，进程在尝试访问新分配的内存区域时出现了问题。用户进程无法访问这些未实际映射的内存区域，导致访问错误。

- **解决办法**：在进行实验时，需要确保 `sbrk()` 函数不仅更新进程的虚拟内存大小，还要正确处理内存映射。在实际操作中，如果要让用户程序正确访问新分配的内存区域，必须确保这些区域已经被正确映射到物理内存。

3. **调试和排查内存错误**：

   - **困难描述**：由于修改后的 `sbrk()` 可能会导致系统崩溃或其他内存错误，排查具体的内存错误可能较为复杂。尤其是在系统崩溃时，找到错误的根源需要细致的调试。
   - **解决办法**：使用调试工具（如 GDB）来跟踪和分析系统崩溃的原因。可以通过查看内存访问和页面映射情况来理解错误的根本原因。此外，查看 `usertrap()` 中的错误处理和调试输出，有助于定位具体的错误点。

这些困难和解决办法说明了内存管理中的一些关键问题，如内存分配的一致性和用户进程的内存访问权限。在实际操作系统开发中，确保虚拟地址空间和物理内存的正确映射对于系统的稳定性和可靠性至关重要。

4) 实验心得

通过这次实验，我深入理解了 `sbrk()` 系统调用如何管理进程的虚拟内存。删除内存分配的代码暴露了虚拟内存管理中的关键问题，特别是如何处理页错误和未分配内存的访问。通过这个实验，我认识到在操作系统中进行内存管理时，确保内存分配和释放的一致性是非常重要的。这对后续更复杂的内存管理任务奠定了基础。

Lazy Allocation (Moderate)

1) 实验目的

本实验的目的是实现惰性内存分配（lazy allocation），以便在用户空间发生页面错误时，内核能够自动分配新的物理页面并映射到出错的虚拟地址。通过修改 `usertrap()` 函数来处理页面错误，并在发生页面错误时动态分配和映射页面，使得系统能够处理未分配内存的访问请求，从而提高内存管理的灵活性。

2) 实验步骤

1. **修改 `usertrap` 以处理页面错误**：

   - **在 `trap.c` 中修改 `usertrap()` 函数来处理页面错误**：

```
void
usertrap(void)
{
    uint64 va;
    struct proc *p = myproc();

    // 检查是否是页面错误
    if (r_scause() == 13 || r_scause() == 15) {
        va = r_stval();

        // 将出错的虚拟地址对齐到页面边界
        va = PGROUNDDOWN(va);

        // 分配新的物理内存页面
        char *mem = kalloc();
        if (mem == 0) {
```

```
            printf("usertrap(): out of memory\n");
            panic("usertrap: out of memory");
        }

        // 映射分配的页面到出错的虚拟地址
        if (mappages(p->pagetable, va, PGSIZE, (uint64)mem, PTE_R |
PTE_W) != 0) {
            printf("usertrap(): mappages failed\n");
            panic("usertrap: mappages failed");
        }

        // 刷新 TLB 并返回用户空间
        sfence_vma();
        return;
    }

    // 处理其他异常
    // ...
}
```

2. **修改 `uvmunmap` 以避免 panic**:

   - **在 `uvmunmap()` 中避免触发 panic**:

```
void
uvmunmap(pagetable_t pagetable, uint64 va, uint64 size, int do_free)
{
    // 解除映射页面
    // ...
    if (do_free) {
        // 释放物理内存页面
        // ...
    }
}
```

3. **编译和运行测试**:

   - **编译并运行 xv6**，执行 `echo hi` 命令。检查是否能够成功执行并产生页面错误。
   - **使用 `vmprint` 函数检查页表内容**，确保页面正确映射。
   - **验证是否有页面错误发生**，并检查系统日志和调试输出，确保惰性内存分配正常工作。

3) 实验中遇到的困难和解决办法

1. **页面分配失败**:

   - **困难描述**：分配新的物理页面时可能会遇到内存不足的问题，导致 `kalloc()` 返回 `0`。
   - **解决办法**：检查系统的内存使用情况，确保足够的物理内存可用。如果 `kalloc()` 失败，需要调试内存管理代码，或增加系统的物理内存。

2. **页面映射失败**:

- **困难描述**：调用 `mappages()` 时可能会遇到映射失败的情况。
- **解决办法**：仔细检查 `mappages()` 的实现，确保页面映射的参数正确，并且没有遗漏必要的权限设置。

3. **调试和排查**：

- **困难描述**：调试页面错误处理和内存分配问题可能较为复杂。
- **解决办法**：使用调试工具（如 GDB）跟踪内存分配和页面映射过程。查看内核日志和 `usertrap()` 的输出，确保错误处理代码正常工作。

4) 实验心得

通过本次实验，我深入理解了如何在操作系统中实现惰性内存分配机制。在处理用户空间页面错误时，动态分配和映射内存页面，提高了系统的灵活性和可靠性。实现这一功能涉及内存管理的多个方面，包括页面分配、映射和解除映射。通过对 xv6 内存管理代码的修改和调试，我进一步提高了对操作系统底层机制的理解，并增强了调试和问题解决的能力。

Lazytests and Usertests (Moderate)

1) 实验目的

本实验的目的是完善惰性内存分配的实现，确保内核能够正确处理各种内存分配相关的异常情况，包括负的 `sbrk()` 参数、页面错误处理、内存不足、以及父子进程的内存复制等。通过修复这些问题，我们能够确保内核的稳定性和可靠性，使得所有的 `lazytests` 和 `usertests` 测试都能通过。

2) 实验步骤

1. **处理负的 `sbrk()` 参数**：

- **在 `sys_sbrk()` 中添加对负参数的处理**：

```
int
sys_sbrk(void)
{
    int n;
    if (argint(0, &n) < 0)
        return -1;

    struct proc *p = myproc();

    if (n < 0 && (p->sz + n < 0))
        return -1;  // Invalid sbrk() call

    p->sz += n;
    return p->sz - n;
}
```

2. **在页面错误处理时处理未分配的虚拟内存地址**：

- **在 `usertrap()` 中添加对未分配地址的处理**：

```
void
usertrap(void)
{
    uint64 va;
    struct proc *p = myproc();

    // 检查是否是页面错误
    if (r_scause() == 13 || r_scause() == 15) {
        va = r_stval();

        // 如果虚拟地址超出了进程分配的地址空间，杀死进程
        if (va >= p->sz || va < USERTOP - PGSIZE) {
            printf("usertrap(): page fault on unallocated address\n");
            exit(-1);
        }

        // 将出错的虚拟地址对齐到页面边界
        va = PGROUNDDOWN(va);

        // 分配新的物理内存页面
        char *mem = kalloc();
        if (mem == 0) {
            printf("usertrap(): out of memory\n");
            exit(-1);
        }

        // 映射分配的页面到出错的虚拟地址
        if (mappages(p->pagetable, va, PGSIZE, (uint64)mem, PTE_R |
PTE_W) != 0) {
            printf("usertrap(): mappages failed\n");
            exit(-1);
        }

        // 刷新 TLB 并返回用户空间
        sfence_vma();
        return;
    }

    // 处理其他异常
    // ...
}
```

3. **处理 `fork()` 时父子进程的内存复制**：
    - **在 `fork()` 中正确处理内存复制**：

```
int
fork(void)
{
    struct proc *np;
```

```
    int i, pid;

    // 省略其他代码...

    // 复制父进程的地址空间
    if (uvmcopy(myproc()->pagetable, np->pagetable, myproc()->sz) < 0)
{
        freeproc(np);
        return -1;
    }

    // 省略其他代码...
    return pid;
}
```

4. **处理未分配地址的系统调用**：

   - **在 `sys_read()` 和 `sys_write()` 中添加对未分配地址的处理**：

```
int
sys_read(void)
{
    char *buf;
    int n;

    if (argint(1, &n) < 0 || argptr(0, &buf, n) < 0)
        return -1;

    // 检查地址是否在用户进程分配的内存范围内
    struct proc *p = myproc();
    if ((uint64)buf + n > p->sz)
        return -1;

    return fileread(myproc()->ofile[0], buf, n);
}

int
sys_write(void)
{
    const char *buf;
    int n;

    if (argint(1, &n) < 0 || argptr(0, &buf, n) < 0)
        return -1;

    // 检查地址是否在用户进程分配的内存范围内
    struct proc *p = myproc();
    if ((uint64)buf + n > p->sz)
        return -1;

    return filewrite(myproc()->ofile[1], buf, n);
}
```

5. **处理内存不足**：

   - **在 `usertrap()` 中处理内存不足的情况**：

```
void
usertrap(void)
{
    // 处理页面错误
    // ...
    if (kalloc() == 0) {
        printf("usertrap(): out of memory\n");
        exit(-1);
    }
    // ...
}
```

6. **处理无效的栈下方页面错误**：

   - **在 `usertrap()` 中处理无效的栈下方页面错误**：

```
void
usertrap(void)
{
    uint64 va;
    struct proc *p = myproc();

    // 检查是否是页面错误
    if (r_scause() == 13 || r_scause() == 15) {
        va = r_stval();

        // 如果地址在用户栈下方，杀死进程
        if (va < USERTOP - PGSIZE) {
            printf("usertrap(): page fault below user stack\n");
            exit(-1);
        }

        // 处理其他页面错误
        // ...
    }
}
```

7. **测试和验证**：

   - **编译并运行 `lazytests`**，确保所有测试通过。
   - **编译并运行 `usertests`**，确保所有测试通过。
   - **检查输出是否符合预期**，验证系统稳定性和内存管理的正确性。

3) 实验中遇到的困难和解决办法

1. **内存分配失败的处理**：
   - **困难描述**：在处理页面错误时，确保 `kalloc()` 失败时能够正确处理。
   - **解决办法**：检查 `usertrap()` 中的内存分配代码，确保在分配失败时能够正确退出进程。
2. **系统调用对未分配地址的处理**：
   - **困难描述**：确保 `sys_read()` 和 `sys_write()` 等系统调用能够正确处理未分配的内存地址。
   - **解决办法**：添加对未分配地址的检查，并在地址无效时返回错误。
3. **父子进程的内存复制**：
   - **困难描述**：在 `fork()` 中，确保父子进程的内存正确复制。
   - **解决办法**：检查 `uvmcopy()` 的实现，确保正确复制内存页表和地址空间。

4) 实验心得

通过本次实验，我深入理解了惰性内存分配的各个方面，特别是在处理各种异常情况时的细节。实现和调试 `lazytests` 和 `usertests`，让我对操作系统的内存管理机制有了更深入的了解，包括页面错误处理、内存分配和系统调用的实现。解决这些问题提高了我的编程能力和问题解决技巧，使我对操作系统的实现有了更全面的认识。