Lab: xv6 lazy page allocation

Git链接：[Jeery1/xv6-6.S081 at lazy](https://github.com/Jeery1/xv6-6.S081/tree/lazy)

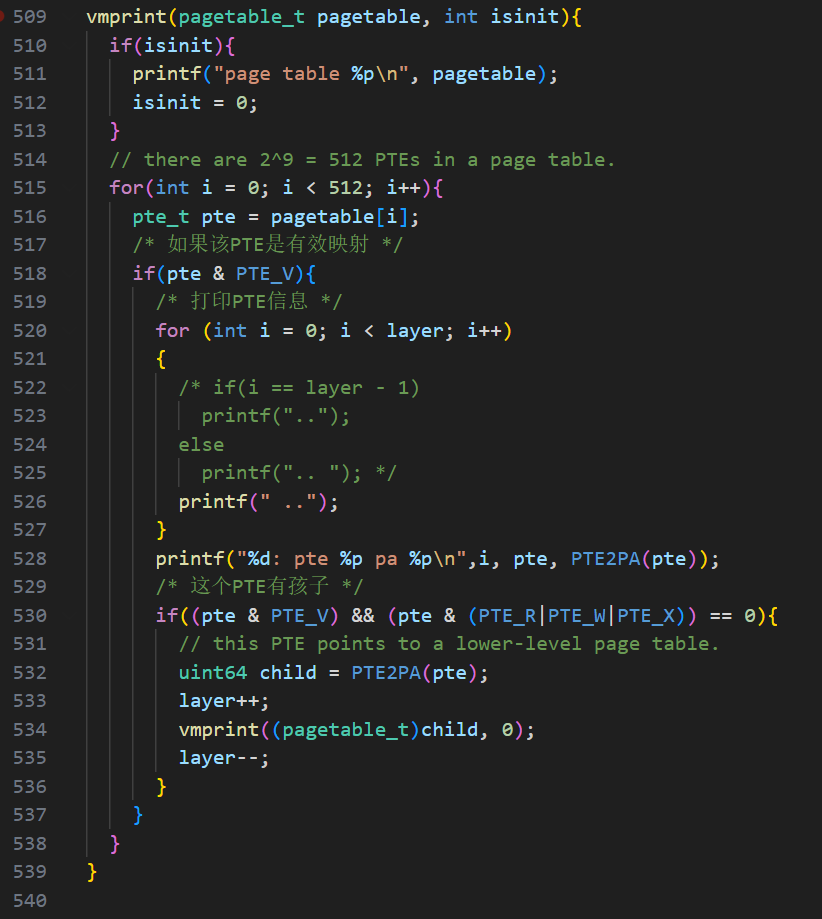
1. 打印页表

**实验目的：**

该实验的目的是实现一个页表打印函数 vmprint，用于调试和深入理解 RISC-V 的三级分页机制。具体要求在 kernel/vm.c 中编写该函数，能够递归遍历并格式化打印给定页表的所有有效映射条目，包括各级页表的索引、页表项（PTE）值及对应的物理地址，并通过缩进（如".."表示顶级页表）清晰展示层级关系。实验需要将该函数集成到 exec.c 中，在创建第一个用户进程时打印其页表，输出格式需与示例一致（如 page table 0x000000007ff6x000 开头，每行显示索引、PTE 和物理地址）。通过调用 riscv.h 中的页表操作宏（如 PTE2PA）并跳过无效条目（PTE\_V=0），该功能将帮助验证内存映射的正确性，为后续内存管理实验（如写时复制）提供调试基础，同时强化对虚拟地址到物理地址转换过程的理解。最终需在 defs.h 声明函数原型以保证跨文件调用。

**实验步骤：**

在vm.c中实现页表打印函数，并且在第一个用户进程中调用它。启动make qemu后就可以看到打印出的页表信息。

代码修改步骤如下：  


Vmprint函数

递归打印页表：遍历给定页表的 512 个 PTE（Page Table Entry），对每个有效（PTE\_V 置位）的 PTE：

打印缩进（.. 表示层级）、索引（i）、PTE 值和物理地址（PTE2PA(pte)）。

若 PTE 是非叶子节点（PTE\_R|PTE\_W|PTE\_X 未设置），递归打印下一级页表。

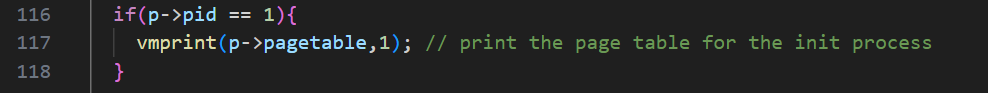
参数说明：

pagetable：当前页表的虚拟地址。

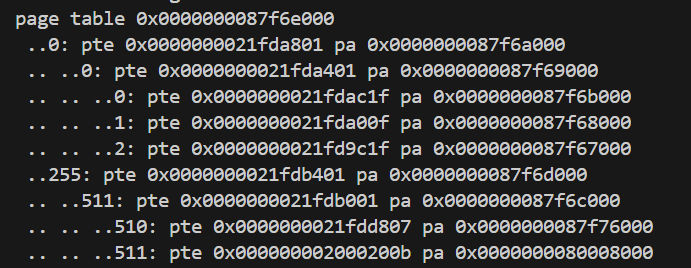
isinit：标记是否为首次调用（打印页表首行地址）。

物理地址转换：PTE2PA(pte) 从 PTE 中提取物理地址。

层级缩进：通过全局变量 layer 控制缩进深度（需在函数外定义 static int layer）。



然后寻找第一个用户进程，在exec.c的exec函数中，调用vmprint

**实验现象：  
**

1. 实现Lazy allocation

**实验目的：**

1. 理解传统内存分配的局限性

问题背景：传统 sbrk() 在用户程序请求堆内存时 立即分配物理页，但实际程序可能不会立即使用全部请求的内存（如预分配大块内存后逐步使用）。

后果：物理内存被无效占用，浪费资源，降低系统整体性能。

2. 实现惰性分配机制

核心思想：

当用户程序调用 sbrk() 时，仅 扩展进程的虚拟地址空间（更新 p->sz），不立即分配物理内存。

实际访问内存触发 页错误（Page Fault） 时，再动态分配物理页并映射。

技术要点：

修改 sys\_sbrk()：取消实际内存分配，仅调整 p->sz。

在 页错误处理逻辑（usertrap） 中捕获缺页异常，按需分配物理页。

3. 解决惰性分配的关键问题

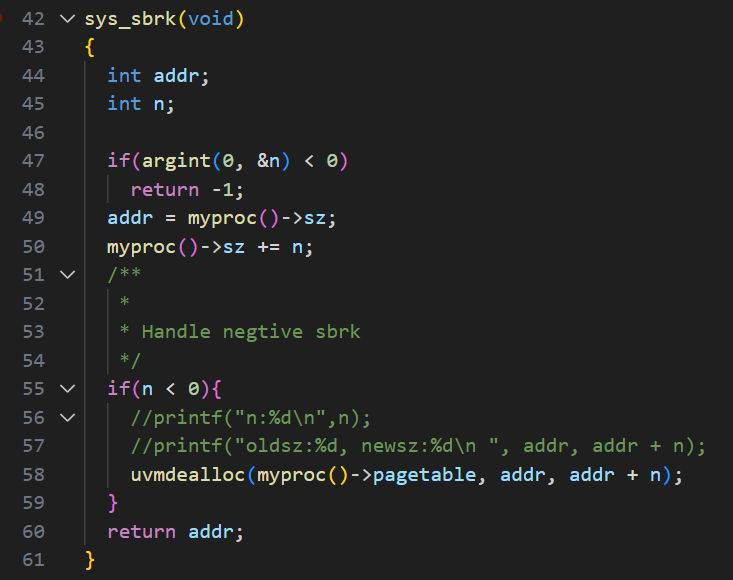
边界检查：确保访问的虚拟地址在合法范围内（va < p->sz 且不进入保护区域如用户栈下方）。

错误处理：非法访问（如超出堆范围）应终止进程，避免安全漏洞。

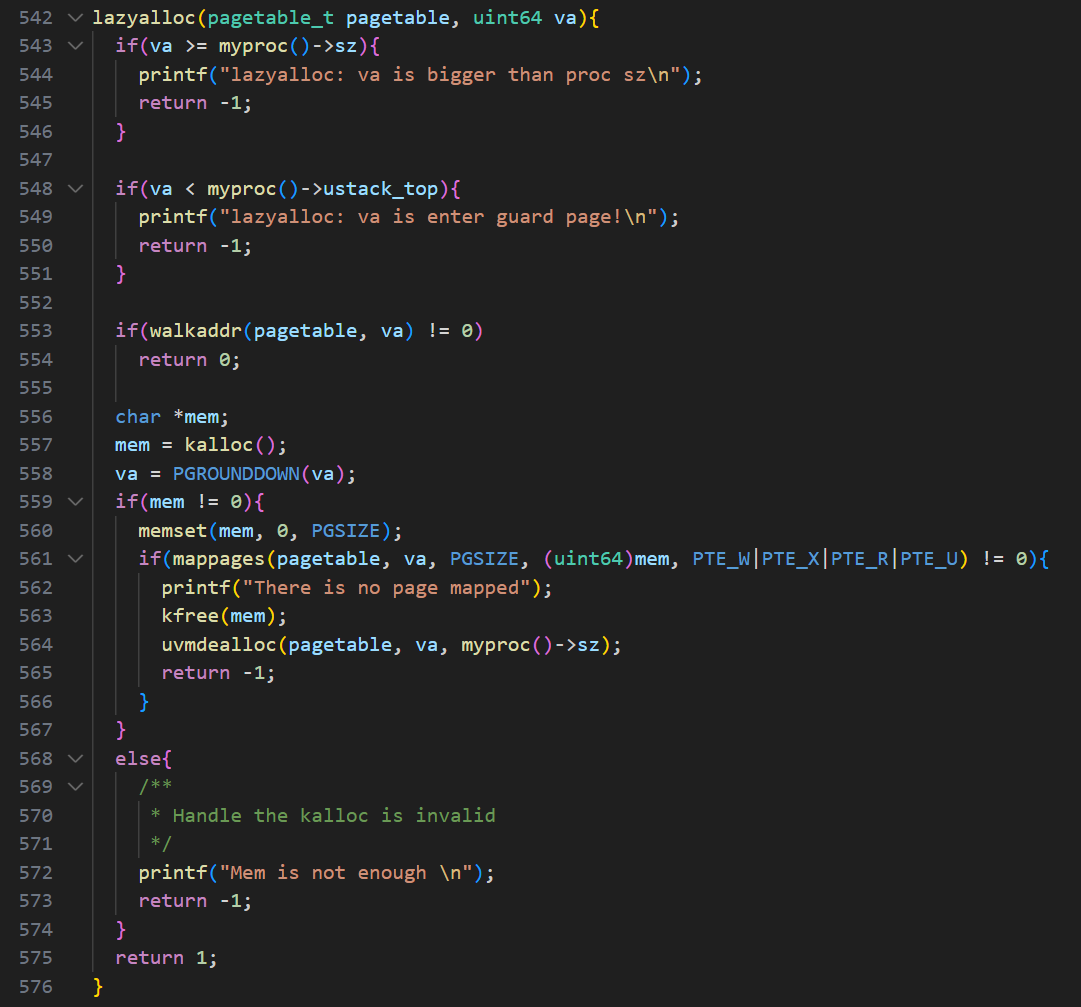
性能优化：减少频繁的页错误处理开销（如预分配连续页面）。

**实验步骤：**

对多个系统文件进行修改，使得系统支持lazy allocation，并能通过lazytests和usertests

修改代码步骤如下：  


找到sysproc.c，删除sys\_sbrk中的growproc()，并改为myproc()->sz += n，这一步的操作是将进程理论大小提高，但暂时并不为其分配内存页面。在处理负数时直接释放内存，确保堆收缩时物理页被回收。



在vm.c中加入lazyalloc函数，实现了惰性分配的核心功能，用于在页错误时动态分配物理内存。

输入检查

检查虚拟地址 va 是否合法：

va >= myproc()->sz：超出堆范围。

va < myproc()->ustack\_top：进入用户栈保护区域（Guard Page）。

若地址无效，打印错误并返回 -1。

页表项检查

通过 walkaddr 检查是否已映射物理页。若已映射，返回 0（无需处理）。

物理页分配与映射

调用 kalloc 分配物理页，对齐地址到页边界（PGROUNDDOWN(va)）。

若分配成功：

清零内存（memset）。

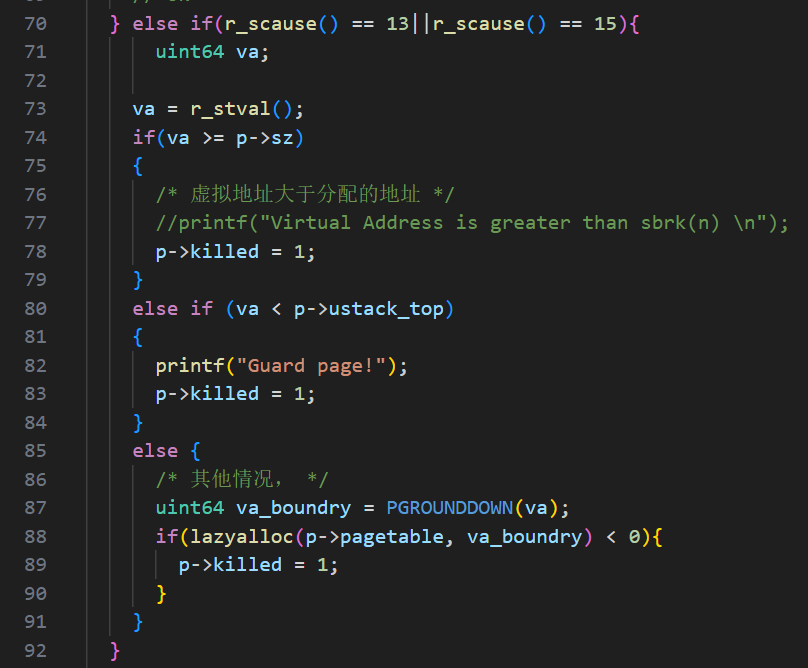
调用 mappages 建立页表映射。

若失败（mem == 0），打印内存不足错误。

错误处理

mappages 失败时，释放物理页并调整堆大小（uvmdealloc）。

权限设置为 PTE\_W|PTE\_X|PTE\_R|PTE\_U（可读/写/执行/用户态访问）。



修改trap.c在出现缺页时不是触发trap，而是尝试利用lazyalloc分配内存。



修改copyout，作用：从内核空间 (src) 复制数据到用户空间 (dstva)。

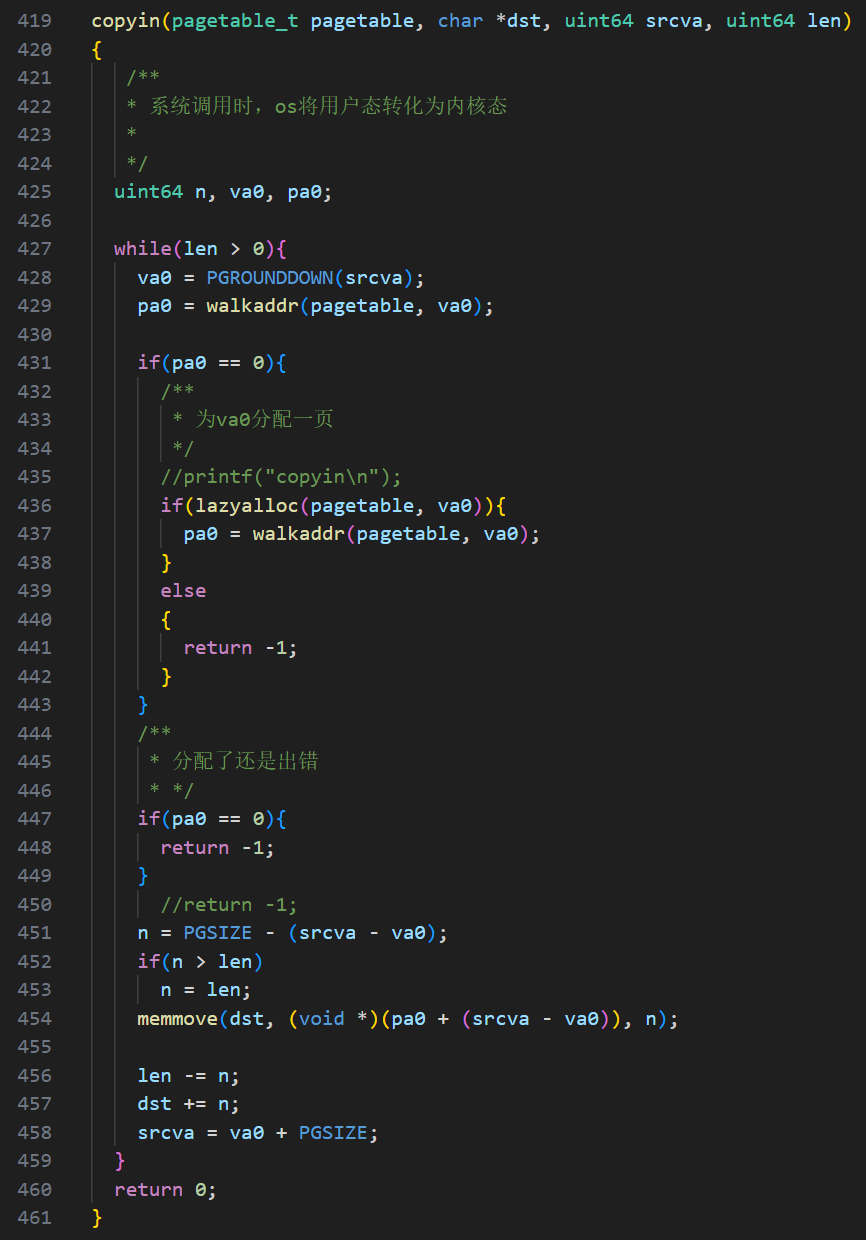
关键步骤：

对齐用户目标地址 dstva 到页边界 (PGROUNDDOWN)。

通过 walkaddr 查找物理地址 pa0。若未映射，调用 lazyalloc 分配物理页。

计算页内偏移量 offset，使用 memmove 复制数据。

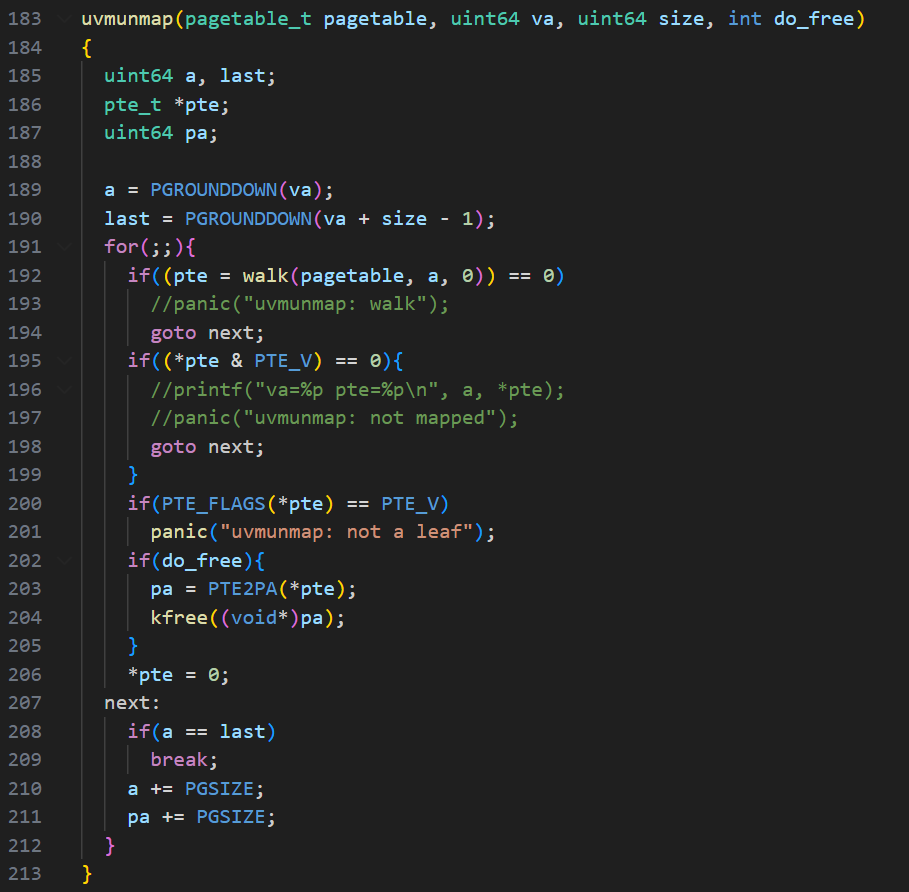
处理跨页情况（len > PGSIZE - offset）。

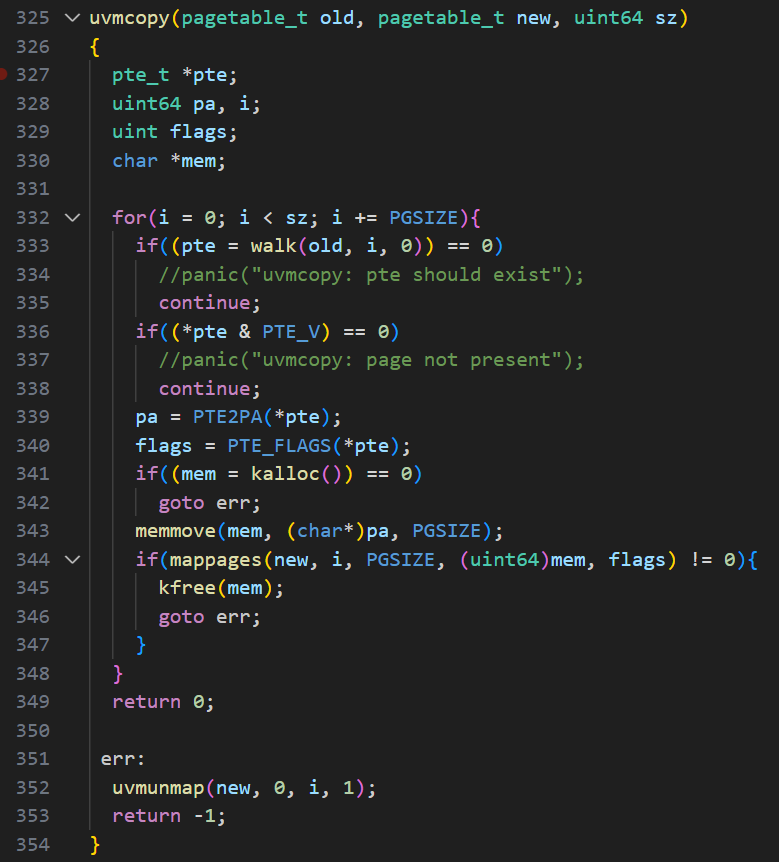


copyin

作用：从用户空间 (srcva) 复制数据到内核空间 (dst)。

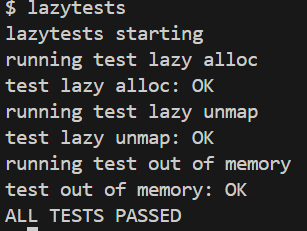
逻辑：与 copyout 对称，仅方向相反。





修改uvmunmap和uvmcopy使得它们不会在发生缺页时触发panic

**实验现象：**



通过lazytests

**为实验报告虚构一段实验中遇到的问题和解决办法：**

问题：sbrk() 系统调用后立即崩溃

问题描述：

修改 sys\_sbrk() 使其仅增长进程的 p->sz 而不分配物理页后，用户程序在首次访问新内存时触发缺页异常，但内核未正确处理，导致 panic。

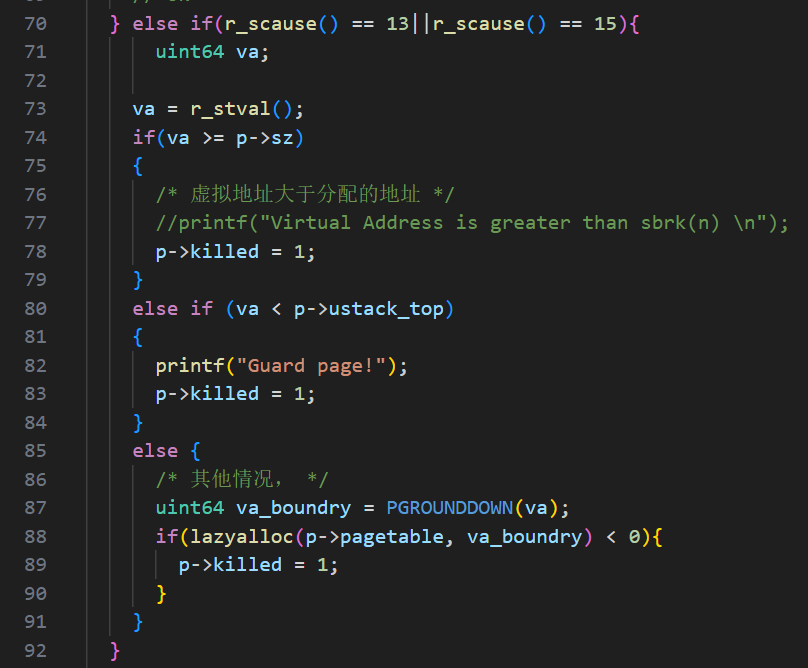
调试过程：

发现崩溃时的 scause 值为 12（Load Page Fault），但未进入预期的缺页处理逻辑。

检查 trap.c 中的 usertrap()，发现未对 r\_scause() 为 12 或 15（Store/Instruction Page Fault）的情况做特殊处理。

解决办法：

在 usertrap() 中添加缺页异常处理逻辑：



问题：用户栈溢出未正确处理

问题描述：

用户程序栈增长时（如递归调用），可能触发惰性分配，但未检查栈溢出，导致内核错误分配物理页。

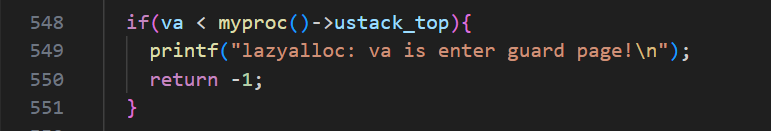
调试过程：

测试递归函数时发现栈指针 sp 向下越界，但仍分配了物理页。

未在缺页处理中校验 va 是否在用户栈的合法范围内。

解决办法：

在 usertrap() 的缺页处理逻辑中加入栈溢出检查：



**实验心得：**

通过本次xv6惰性内存分配实验，我深刻理解了现代操作系统中按需分页机制的精妙设计。在实现过程中，最让我印象深刻的是虚拟内存与实际物理内存分离带来的灵活性——通过延迟分配策略，系统既能快速响应用户内存请求，又能避免不必要的资源浪费。调试缺页异常处理时，我意识到内存管理必须与进程控制紧密配合，特别是在fork()和栈增长等场景中，任何边界条件疏漏都可能导致严重错误。这次实验不仅让我掌握了缺页中断处理、地址合法性校验等关键技术点，更让我体会到操作系统设计中的权衡艺术：在性能优化、资源利用和安全防护之间找到平衡点。这些经验对我后续学习更复杂的内存管理机制（如写时复制、页面置换等）奠定了坚实基础。