

Lab2 system calls

前置知识

页表

1. 地址空间隔离

例子： 假设我们有两个进程A和B，它们分别有自己的虚拟地址空间。进程A可能在其虚拟地址空间中有一个指向地址 `0x1000` 的指针，而进程B也可能在它的虚拟地址空间中有一个指向同样 `0x1000` 的指针。然而，通过页表机制，这两个虚拟地址 `0x1000` 可以分别映射到不同的物理内存地址，例如 `0x2000` 和 `0x3000`，从而确保进程A和B互不干扰。

作用： 通过这种方式，操作系统可以确保每个进程有自己的独立地址空间，从而避免一个进程访问或修改另一个进程的内存数据。这种隔离也有助于提高系统的安全性和稳定性。

2. 共享内存的实现

例子： 假设进程A和进程B需要共享一块内存来进行数据交换。操作系统可以在两个进程的页表中将相同的虚拟地址映射到相同的物理内存地址上。例如，进程A的虚拟地址 `0x4000` 和进程B的虚拟地址 `0x5000` 都可以映射到物理地址 `0x6000`，从而实现共享内存。

作用： 这种技巧允许多个进程共享数据，同时每个进程仍然可以有自己的独立的虚拟地址空间。这在需要进程间通信的场景中非常有用。

3. 内核和用户空间的区分

例子： 在xv6中，所有的用户进程的内存访问都是通过页表管理的。用户空间和内核空间是通过不同的页表映射来区分的。例如，用户进程的页表只允许它们访问自己的代码和数据，而不允许它们直接访问内核的代码和数据。通过将内核代码映射到用户进程的页表中一个特定的地址空间（通常在高地址部分），同时设置访问权限（只读或不可访问），xv6可以有效地保护内核内存不被用户进程修改。

作用：通过这种方式，操作系统可以保护内核不受用户进程的干扰，确保系统的稳定性和安全性。这也是用户模式和内核模式隔离的基础。

4. 栈保护

例子：为了保护用户进程的栈，操作系统可以在栈的末尾映射一个不可访问的页。如果用户程序试图访问超出栈的有效范围的地址，访问将导致页错误，从而防止栈溢出攻击。

作用：这种页表技巧通过设置一个没有映射的页（即不可访问的页）来防止栈溢出，增强了系统的安全性。栈溢出是常见的安全漏洞，通过这种方式可以有效地防止栈溢出导致的恶意代码执行。

总结

通过页表，操作系统可以实现虚拟内存管理，确保进程之间的隔离，支持内存共享，实现内核和用户空间的分离，并通过一些技巧保护内存安全。每一个这些例子都展示了页表在现代操作系统中的核心作用，以及它们如何通过映射不同的虚拟地址来实现复杂的内存管理功能。

分页硬件

地址转换过程和原理

在 x86 架构中，虚拟地址和物理地址之间的转换是通过页表（Page Table）和分页硬件（paging hardware）实现的。下面是一个具体的例子来说明这个过程：

例子：

假设有一个用户程序访问了虚拟地址 `0x12345678`，操作系统需要将这个虚拟地址转换为实际的物理地址，以便读取或写入物理内存。

1. 虚拟地址的拆分：

- 虚拟地址 `0x12345678` 在二进制下表示为 `0001 0010 0011 0100 0101 0110 0111 1000`。
- 高 10 位 `0001 0010 00` 用于在页目录（Page Directory）中找到对应的页目录条目（PDE）。

- 中间 10 位 **0011 0100 01** 用于在选定的页表页（Page Table）中找到具体的页表条目（PTE）。
- 最后 12 位 **0101 0110 0111 1000** 表示页内偏移，用于定位页内具体地址。

2. 页目录查找：

- 使用虚拟地址的高 10 位（**0001 0010 00**）在页目录中查找对应的页目录条目（PDE）。
- 该 PDE 指向一个页表页（Page Table Page），其中包含多个页表条目（PTE）。

3. 页表查找：

- 使用虚拟地址的中间 10 位（**0011 0100 01**）在选定的页表页中查找对应的页表条目（PTE）。
- 该 PTE 包含指向物理页框的物理页号（PPN）以及一些控制标志位。

4. 物理地址的生成：

- 将 PTE 中的物理页号（PPN）替换虚拟地址的高 20 位。
- 保留虚拟地址的最后 12 位不变，这部分是页内偏移。
- 这样，虚拟地址 **0x12345678** 被转换为物理地址，例如 **0x56789078**。

PTE、PPN 和 PDE 的含义及关系

- **PTE（Page Table Entry）页表条目**：每个 PTE 包含指向物理内存的物理页号（PPN）以及一些控制标志位，如 PTE_P、PTE_W、PTE_U 等。这些标志位控制页的权限和状态。
- **PPN（Physical Page Number）物理页号**：PPN 是物理页在物理内存中的编号，它与页内偏移相结合，最终形成物理地址。
- **PDE（Page Directory Entry）页目录条目**：PDE 是页目录中的一个条目，指向一个页表页（Page Table Page）。它用虚拟地址的高 10 位进行索引，找到对应的页表页，然后页表页中的 PTE 再指向具体的物理页。

PTE、PPN 和 PDE 的关系

在虚拟地址转换过程中，PDE、PTE 和 PPN 之间的关系如下：

1. 页目录查找 (PDE)：

- 虚拟地址的高 10 位用于在页目录中查找对应的 PDE。
- PDE 指向一个具体的页表页，页表页中包含多个 PTE。

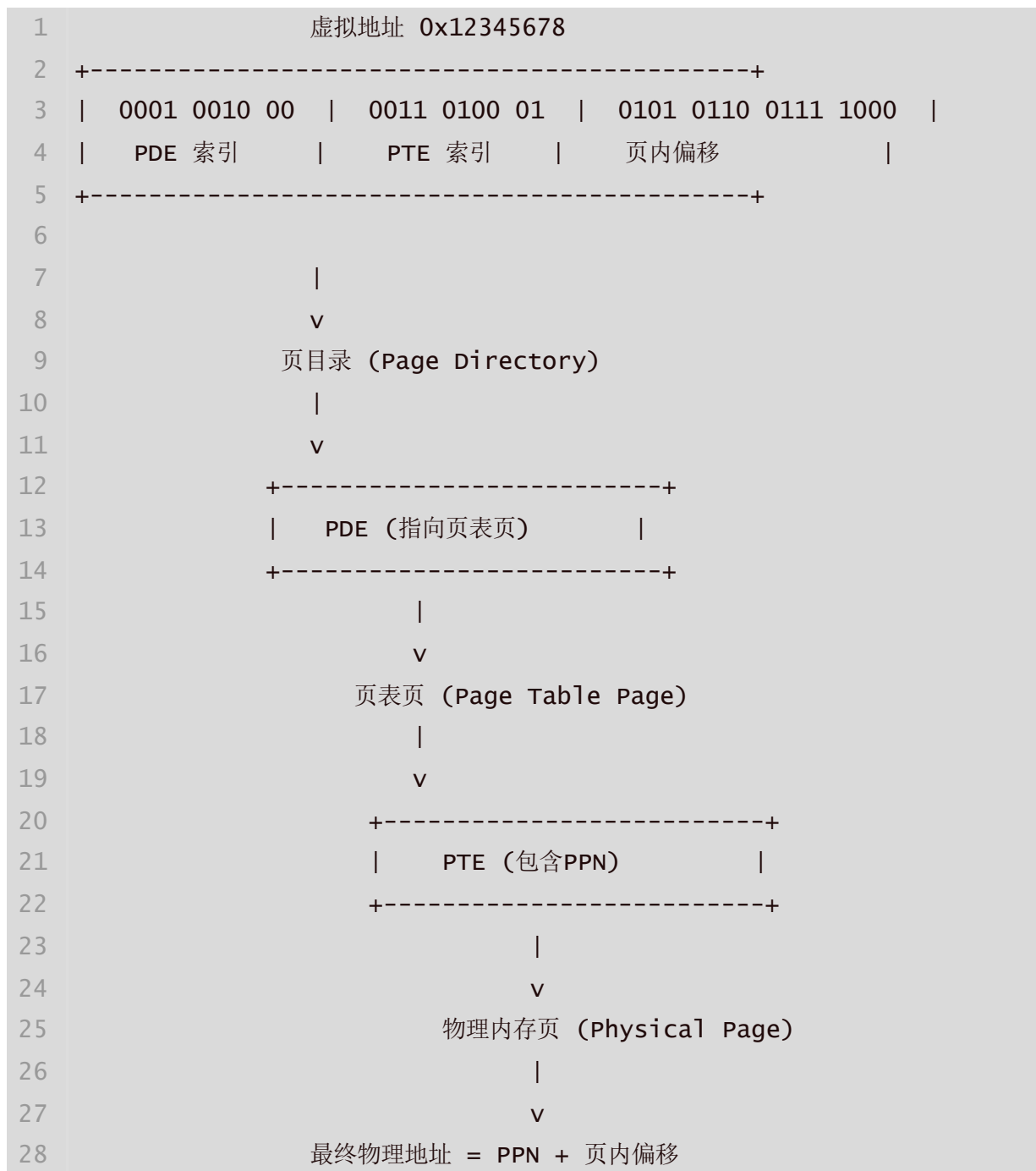
2. 页表查找 (PTE):

- 虚拟地址的中间 10 位用于在选定的页表页中查找对应的 PTE。
- PTE 包含指向物理内存的物理页号 (PPN) 以及相关的控制标志位。

3. 物理地址生成 (PPN):

- PPN 是 PTE 中保存的物理页号，将其替换虚拟地址的高 20 位。
- 虚拟地址的低 12 位作为页内偏移，与 PPN 结合生成最终的物理地址。

示意图



分页硬件

分页硬件（**paging hardware**）是处理器中的一个组件，它负责根据页表将虚拟地址转换为物理地址。它会自动完成前述的页目录和页表查找，以及地址的转换。这些操作对程序员和用户程序是透明的，即用户程序只需使用虚拟地址，而不需要关心底层的地址转换过程。

分页硬件还负责管理页表中的标志位，例如控制某个页是否有效、是否允许写操作、是否只允许内核访问等。这些标志位可以保护内存，确保用户程序不能随意访问或修改内核数据或其他进程的数据。

例子总结和技巧

- **多进程隔离**：每个进程有自己独立的页表，使得进程之间的地址空间相互隔离，避免一个进程访问另一个进程的数据。
- **共享内存**：不同的进程可以通过页表将相同的虚拟地址映射到相同的物理内存页，从而实现进程间的数据共享。
- **内核保护**：操作系统通过设置页表中的标志位，防止用户进程访问内核内存，确保系统的安全性和稳定性。
- **栈保护**：在栈底部设置一个不可访问的页面（通过PTE的标志位），防止栈溢出攻击。

这些机制和技巧确保了操作系统的稳健性、安全性，以及对内存的高效管理。

进程地址空间

entry 与 kvmalloc 的映射差异

在 **entry** 阶段，系统刚刚启动，页表已经建立了一些基本的映射，使得内核的 C 代码可以正常运行。然而，这些映射只是最低限度的映射，主要是为了确保系统可以启动并执行基础代码。接下来，**main** 调用了 **kvmalloc** 来切换到一个更复杂和完整的页表，这个页表对内存空间进行了更精细的映射。

进程和页表的关系

- **进程的用户内存**：每个进程都有自己独立的页表，其用户内存从虚拟地址 **0** 开始，最多可以增长到 **KERNBASE**（即 2GB）。这样，进程能够在这个范围内申请内存并使用。

- **内核内存的映射：** 在每个进程的页表中，除了映射用户内存外，还包含了内核运行所需要的映射。这些映射都在 `KERNBASE` 之上。这意味着即使进程从用户态切换到内核态（例如通过中断或系统调用），也不需要切换页表，因为内核所需的内存已经在进程的页表中进行了映射。

映射的原因

文中提到，内核将虚拟地址 `KERNBASE:KERNBASE+PHYSTOP` 映射到物理地址 `0:PHYSTOP` 的原因有以下几点：

1. **内核使用自己的指令和数据：** 内核代码需要访问自己的数据和执行自己的指令，这些代码和数据存储在物理内存中。通过将这段虚拟地址映射到对应的物理地址，内核可以轻松访问这些资源。
2. **内核操作物理页：** 在系统中，内核需要直接操作物理内存，特别是在创建页表页时。将 `KERNBASE` 以上的虚拟地址映射到实际的物理内存可以让内核更方便地访问和操作这些物理页。

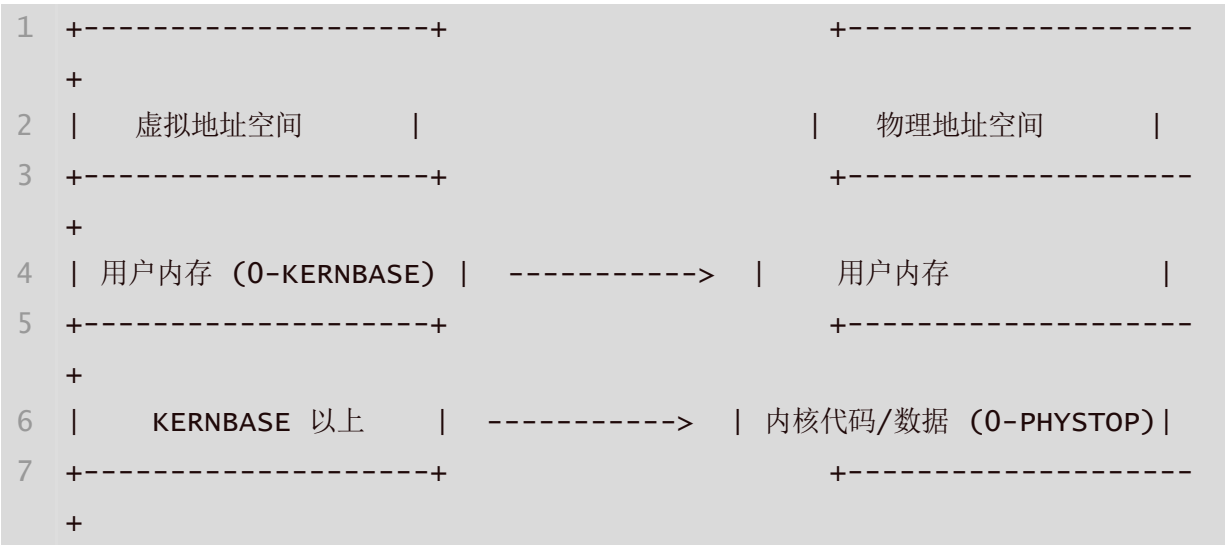
映射的限制与不足

这种映射方式的一个主要限制是，xv6 无法使用超过 2GB 的物理内存。这是因为内核的虚拟地址空间被限制在 `KERNBASE` 以上，而这个范围内的地址空间只能映射到 2GB 的物理内存。此外，对于一些使用高地址（如 `0xFE000000` 以上）的 I/O 设备，xv6 采用了直接映射的方式来确保这些设备能够正常工作。

内核和用户内存的映射关系

每个进程的页表同时包括用户内存和内核内存的映射。这意味着当用户态切换到内核态时，不需要进行页表的转换，从而提高了系统的效率。此外，内核并没有独立的页表，而是借用了用户进程的页表来运行。

示意图



什么是 entry?

entry 是指系统在启动时的最初阶段，它是内核代码的起点。在这个阶段，操作系统从裸机状态（即没有操作系统运行的状态）逐步建立起最基本的系统结构，例如设置页表、初始化硬件设备等，使得后续的复杂操作能够顺利进行。在 **entry** 阶段，系统会设置一些简单的映射和初始化工作，以便能够顺利运行内核的 C 代码。

内核操作物理页

在操作系统中，内核需要直接管理和操作物理内存，例如在创建和管理页表时。因此，内核必须能够访问系统的物理内存。由于在现代操作系统中，内核和用户程序使用的是不同的虚拟地址空间，内核必须通过特定的映射机制来操作物理内存。

将 KERNBASE 以上的虚拟地址映射到实际的物理内存

在 x86 架构的 xv6 操作系统中，内核的代码和数据通常位于 **KERNBASE** 以上的虚拟地址空间。**KERNBASE** 是一个虚拟地址，它对应的是物理地址的开始（例如物理内存的 **0x0** 地址）。通过将 **KERNBASE** 以上的虚拟地址映射到物理地址 **0x0** 开始的位置，内核可以方便地访问物理内存中的每一页。

例子：创建页表页

当内核需要创建新的页表页时，必须为该页表页分配物理内存。这个过程可以分为以下几个步骤：

1. **分配物理页：** 内核需要在物理内存中找到一个空闲的页，准备将它用作新的页表页。这一步通常通过一个内存分配器来完成，例如 `kalloc()` 函数。
2. **设置页表条目（PTE）：** 内核会在当前进程的页表中为新的页表页设置一个页表条目（PTE）。这个 PTE 的主要作用是将虚拟地址映射到刚刚分配的物理页。
3. **映射虚拟地址到物理地址：** 内核通过设置 PTE 中的 PPN（物理页号）和其他标志位，将虚拟地址（例如 `KERNBASE` 以上的地址）映射到实际的物理页。例如，如果内核分配了一个物理页号为 `0x200` 的物理页，那么可以将虚拟地址 `KERNBASE + 0x200` 映射到这个物理页。
4. **访问物理页：** 现在，内核就可以通过虚拟地址（如 `KERNBASE + 0x200`）访问物理内存中的该页。这使得内核可以在需要时直接操作物理页，而不必通过复杂的地址转换过程。

示意图

假设内核需要分配和操作一个物理页，地址为 `0x20000`。通过将虚拟地址 `KERNBASE + 0x20000` 映射到物理地址 `0x20000`，内核可以方便地使用该虚拟地址来访问和操作这个物理页。

```
1  +-----+ 映射  +-----+
2  | 虚拟地址: KERNBASE+0x20000 | ----> | 物理地址: 0x20000 |
3  +-----+          +-----+
4
5  // 内核通过虚拟地址 KERNBASE+0x20000 直接访问物理地址 0x20000 对应的物理页。
```

代码：建立一个地址空间

函数作用与调用过程说明

在 `xv6` 操作系统中，内核需要建立页表以管理虚拟地址到物理地址的映射。以下是 `main` 函数、`kvmalloc`、`setupkvm`、`mappages` 和 `walkpgdir` 函数的详细说明，以及它们的调用过程和作用点。

1. `main` 函数

作用：`main` 函数是操作系统启动的入口点。它负责启动和初始化操作系统，包括设置内核的页表、初始化硬件和启动第一个用户进程。

调用过程和作用点：

- 在 `main` 函数中，操作系统需要切换到一个拥有内核运行所需的虚拟地址到物理地址映射的页表。为此，`main` 调用了 `kvmmalloc` 函数，创建一个新的页表，并切换到这个新的页表中。

2. `kvmmalloc` 函数

作用：`kvmmalloc` 函数用于创建并切换到一个新的页表，该页表包含内核运行所需的虚拟地址到物理地址的映射。它通过调用 `setupkvm` 来完成大部分工作。

调用过程和作用点：

- `kvmmalloc` 首先调用 `setupkvm` 来创建一个新的页表，`setupkvm` 函数负责为内核建立必要的虚拟地址映射。
- 创建完新的页表后，`kvmmalloc` 切换到这个新的页表，使其成为当前活跃的页表。

3. `setupkvm` 函数

作用：`setupkvm` 函数负责初始化一个新的页表，并为内核创建一组基本的虚拟地址到物理地址的映射。这些映射包括内核指令和数据所需的内存、物理内存（`PHYSTOP` 以下）以及 I/O 设备所需的内存。

调用过程和作用点：

- `setupkvm` 首先分配一页内存，用于存放页目录。
- 然后，它调用 `mappages` 函数，为内核建立所需的映射，这些映射信息存储在 `kmap` 数组中。
- `setupkvm` 只建立内核需要的映射，不会处理用户内存的映射，这些映射在稍后的进程加载时处理。

4. `mappings` 函数

作用：`mappings` 函数用于在页表中建立从一段虚拟内存到一段物理内存的映射。它逐页处理，将每个虚拟地址映射到相应的物理地址。

调用过程和作用点：

- `mappings` 函数逐页处理待映射的虚拟内存地址。
- 对于每个虚拟地址，它调用 `walkpgdir` 函数，找到对应的 PTE（页表条目）地址。
- 然后，它初始化该 PTE，将物理页号、权限标志（如 `PTE_W`、`PTE_U`）以及 `PTE_P` 位（表示该页是否有效）写入 PTE。

5. `walkpgdir` 函数

作用：`walkpgdir` 函数模仿 x86 的分页硬件，查找一个虚拟地址对应的 PTE。它用于在页表中找到给定虚拟地址的页表条目（PTE）。

调用过程和作用点：

- `walkpgdir` 使用虚拟地址的前 10 位找到页目录中的对应条目。
- 如果该条目不存在并且 `alloc` 参数被设置为 1，它将分配一个新的页表页，并将其物理地址放入页目录。
- 最后，使用虚拟地址的接下来的 10 位找到页表中的 PTE 地址。

调用关系示意图

```
1  main
2  |— kvmalloc
3  |   └─ setupkvm
4  |       └─ mappings
5  |           └─ walkpgdir
6  |               └─ kmap (映射数组)
7  └─ ...
```

示例与解释

- **页表的建立与切换：** 在系统启动时，`main` 函数调用 `kvmalloc`，创建并切换到一个新的页表。该页表包含了内核指令和数据、物理内存、I/O 设备的映射，这些映射由 `setupkvm` 通过调用 `mappages` 函数逐步建立。
- **地址映射过程：** 当内核需要访问某个虚拟地址（如 `KERNBASE + 0x20000`），它首先通过 `walkpgdir` 在页目录中查找对应的 PTE。如果找到了对应的 PTE，内核就可以通过该 PTE 将虚拟地址转换为物理地址，进而访问实际的物理内存。

总结

这些函数的组合实现了操作系统中的虚拟内存管理机制。通过对页表的管理和操作，系统可以将虚拟地址映射到实际的物理内存，并为每个进程提供独立的地址空间。这不仅提高了系统的安全性，还简化了内存管理和资源分配。

物理内存的分配和自举问题

在操作系统运行时，内核需要动态分配物理内存，用于存储页表、进程的用户内存、内核栈以及管道缓冲区等数据结构。xv6 操作系统使用从内核结尾到 `PHYSTOP` 之间的物理内存作为运行时的内存资源。每次分配内存时，xv6 会分配一整块大小为 4096 字节的页，并通过维护一个物理页链表来管理和分配空闲页。这个链表记录了所有空闲的物理页，当需要分配内存时，系统会将一个空闲页从链表中移除，而当需要释放内存时，系统会将该页重新加入链表。

自举问题

在这个上下文中，我们遇到了一个典型的自举问题（bootstrap problem），这个问题可以简单理解为：在系统初始化的早期阶段，为了进行某项操作，系统需要依赖其自身尚未完全初始化的部分。

具体问题：

- **内存分配的依赖关系：** 要初始化内存分配器，系统需要创建一个包含所有物理内存页的链表。这个链表需要用页表来管理，而页表本身也需要存储在物理内存中。因此，页表的创建和内存分配存在一个相互依赖的关系——要建立页表需要分配物理内存，而要分配物理内存又需要依赖页表。

解决方法

xv6 通过在系统的初始化阶段使用一个特殊的页分配器来解决这个问题。这个分配器在系统的最早期——内核初始化阶段——进行使用，它的工作原理如下：

1. **特殊的页分配器**：这个分配器直接从内核数据段的末尾开始分配内存，不依赖于标准的页表机制。这意味着它可以在页表尚未完全建立之前使用。
2. **限制**：该分配器不支持释放内存，并且只能分配有限的内存（4MB），这是因为它仅依赖于 `entrypgdir` 页表的映射范围。但这个内存量足以支持内核完成最初的页表创建和系统初始化工作。
3. **完整页表的建立**：一旦内核的初始页表建立起来，系统就可以切换到标准的页表机制，并通过标准的页分配器来管理内存。

自举问题的本质

自举问题的本质是系统初始化的互相依赖关系。在这个例子中，为了管理内存需要页表，而为了创建页表需要内存。xv6 通过在系统的早期阶段使用一个特别设计的简化分配器，绕过了这种依赖，从而成功完成系统的初始化。

代码：物理内存分配器

1. 数据结构：

- 内存分配器使用一个空闲链表来管理可分配的物理内存页。这个链表的每个元素是一个 `struct run` 结构体。
- `struct run` 结构体保存了链表的下一个节点的指针，即 `r->next`，形成一个链表。

2. 内存获取：

- 分配器将 `struct run` 结构体直接存放在空闲页的内存中，因为空闲页没有其他数据，这样每个空闲页本身就是链表中的一个节点。

3. 锁机制：

- 分配器使用一个 `spin lock` 来保护这个空闲链表，以避免在多核环境下并发访问时发生冲突。锁和链表都封装在一个结构体中，确保锁保护该结构体的所有成员。

4. 初始化：

- `main` 函数通过调用 `kinit1` 和 `kinit2` 两个函数来初始化内存分配器。
- `kinit1` 用于初始化不超过 4MB 的内存，这部分内存不需要使用锁进行保护。

- `kinit2` 在初始化后期被调用，允许使用锁，并扩展到更多的内存可用于分配。
- 由于在 x86 架构上很难准确检测出机器的物理内存量，`xv6` 假设机器有 240MB 的物理内存，并使用 `PHYSTOP` 作为内存的上限。

5. 内存加入空闲链表：

- `kinit1` 和 `kinit2` 调用 `freerange` 函数将指定范围内的内存加入空闲链表中。
- `freerange` 调用 `kfree` 来释放每一页内存并将其加入空闲链表。

6. 内存对齐：

- 由于页表条目（PTE）只能指向 4096 字节对齐的物理地址（即地址必须是 4096 的倍数），`freerange` 使用 `PGROUNDUP` 来确保分配器释放的内存地址是对齐的。

7. 虚拟地址与物理地址：

- 分配器通过映射到高内存区域的虚拟地址找到对应的物理页，而非直接使用物理地址。这是因为内核的地址空间在启动时是以高虚拟地址开始的。
- `kinit` 使用 `p2v(PHYSTOP)` 将物理地址 `PHYSTOP` 转换为对应的虚拟地址。

8. 地址处理：

- 分配器中的地址有时被视为整数进行运算（例如在 `kinit` 中遍历所有页），有时被视为指向内存的指针（例如操作页中的 `struct run`）。这种双重用法导致代码中存在大量的类型转换。

9. 释放内存：

- `kfree` 函数首先将释放的内存页中的每一字节设为 1，以防止程序错误地访问已释放的内存，确保这样的错误能早期暴露。
- 然后 `kfree` 将内存地址 `v` 转换为一个指向 `struct run` 的指针，并将这个新释放的页插入到空闲链表的头部。

10. 分配内存：

- `kalloc` 函数从空闲链表中移除并返回链表的表头，这意味着它返回了一个已分配的物理页，并从空闲链表中删除该页。

xv6 地址空间中的用户部分

在 `xv6` 操作系统中，地址空间被划分为用户部分和内核部分。用户部分包含了运行用户进程所需的所有内存，包括代码、数据、堆、栈等。下面详细介绍 `xv6` 地址空间中用户部分的结构和特点。

1. 用户内存结构

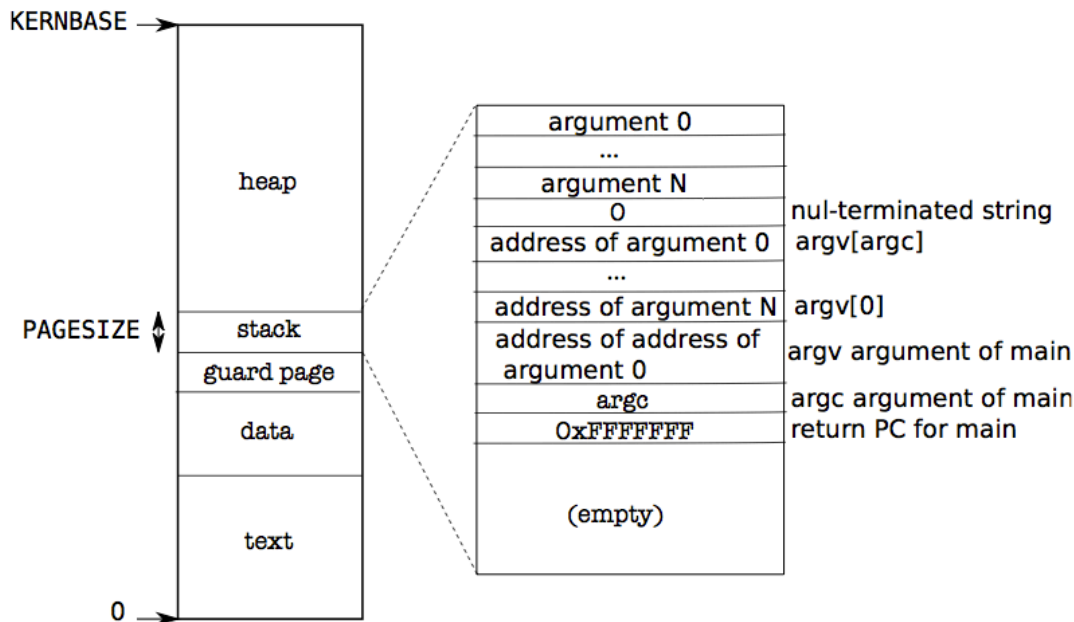


Figure 2-3. Memory layout of a user process with its initial stack.

- 图表 2-3 展示了一个正在运行的进程在 xv6 中的用户内存结构。这个内存结构通常从较低的虚拟地址开始增长，并且包括以下几个关键部分：
 - **代码段和数据段**：代码段包含可执行程序的指令，数据段包含静态数据。这些段位于用户地址空间的底部。
 - **堆（Heap）**：堆用于动态内存分配，通常位于数据段和栈之间。堆可以通过 `sbrk` 系统调用动态增长，以适应程序运行时的内存需求。
 - **栈（Stack）**：栈是用于函数调用和局部变量存储的区域。在 xv6 中，每个用户进程的栈通常占用一页内存。栈从高地址向低地址增长。

2. 栈的布局

- 栈的顶部通常保存程序的命令行参数和指向这些参数的指针数组。
- 紧接着，栈中存放了程序入口函数 `main(argc, argv)` 所需的初始值，这些值使得程序可以像刚刚调用 `main` 函数一样开始执行。

3. 保护页

- 在栈的下方，xv6 引入了一个保护页（**guard page**）。这个保护页没有映射到任何物理内存，即它不对应于实际的物理内存。

- **目的：**保护页的主要作用是防止栈意外地越界使用内存。如果栈增长超过其分配的一页内存，程序将试图访问保护页对应的地址。由于该地址没有映射到物理内存，因此访问时会触发异常。这种机制可以帮助程序在越界访问时快速失败，而不是导致更严重的错误。

4. 堆的增长

- xv6 允许用户程序通过 `sbrk` 系统调用来扩展堆的大小。堆的增长方向是向上，即从低地址向高地址增长。堆的增长可能会接近栈的区域，因此需要合理管理堆和栈之间的内存使用，以防止两者发生冲突。

exec 创建地址空间中用户部分的过程

1. 打开二进制文件：

- `exec` 通过 `namei` 函数打开指定的二进制文件，该函数用于查找文件系统中的文件。在第 6 章会详细解释 `namei` 的工作原理。

2. 读取 ELF 头：

- xv6 应用程序使用 ELF 格式来描述，这种格式在 `elf.h` 中定义。一个 ELF 文件包括一个 ELF 头（`struct elfhdr`）和若干个程序段头（`struct proghdr`）。
- `exec` 的第一步是检查文件是否包含 ELF 二进制代码。它通过验证 ELF 头中的魔法数字（`ELF_MAGIC`）来判断该文件是否为合法的 ELF 格式。

3. 分配新的页表：

- 通过 `setupkvm` 函数，`exec` 分配了一个新的页表，但此时没有任何用户部分的映射。新的页表将用于管理进程的内存映像。

4. 分配和加载内存：

- 对于每个 ELF 程序段，`exec` 调用 `allocuvm` 为其分配足够的虚拟内存空间。`allocuvm` 会检查所分配的虚拟地址是否低于 `KERNBASE`，确保这些内存属于用户空间。
- 然后，`exec` 使用 `loaduvm` 函数将程序段内容从文件中加载到分配的内存中。`loaduvm` 使用 `walkpgdir` 来找到虚拟地址对应的物理地址，并通过 `readi` 函数读取文件内容到内存中。

5. 初始化用户栈：

- `exec` 为用户栈分配一页内存，并将程序的命令行参数拷贝到栈顶。它将指向这些参数的指针保存在 `ustack` 中，并在 `argv` 列表的最后放置一个空指针。

- 在栈的下一页（即栈下方）设置一个无法访问的保护页，以防止栈溢出。当栈增长超过一页时，访问这个页会引发异常，从而保护内存不被非法访问。

6. 处理错误：

- 在创建新内存映像的过程中，如果 `exec` 发现了错误（如无效的程序段），它会跳转到 `bad` 标签处，释放已分配的内存，并返回 `-1` 以表示执行失败。
- `exec` 在确认一切都成功后才会释放旧的内存映像，以避免在发生错误时无法返回 `-1`。

7. 加载新映像并释放旧映像：

- 当所有的内存映像和数据都成功加载后，`exec` 将新映像装载到进程中，并释放旧的内存映像，完成内存的切换。

8. 返回成功：

- 最后，`exec` 成功返回 `0`，表示系统调用执行成功，并且进程的内存映像已更新。

关于xv6

以下是xv6的一些特性总结：

1. 基础分页硬件支持：

- xv6 使用分页硬件来保护和映射内存。虽然它与许多现代操作系统一样使用分页技术，但其实现较为简单，缺乏某些高级功能。

2. 简化的内存管理：

- xv6 不支持一些高级内存管理技术，如从磁盘请求页、写时复制（`copy-on-write`）操作、共享内存和惰性分配页（`lazily-allocated page`）。
- xv6 也不支持自动扩展栈，当栈溢出时不会自动增加栈的大小。

3. 有限的段式内存转换支持：

- x86 支持段式内存转换，但 xv6 只利用段式内存来实现每个CPU的固定地址变量（`per-CPU`变量），即 `proc` 结构体。段式内存转换允许不同的CPU在相同的地址上有不同的值。对于不支持段式内存的体系结构，通常需要一个额外的寄存器来指向每个CPU的数据区域。

4. 使用“超级页”优化内存管理：

- 在内存充足的机器上，xv6 使用4MB大小的“超级页”来减少页表的开销。xv6 在初始页表中使用了超级页，通过设置 `%cr4` 寄存器中的 `CP_PSE` 位来通知分页硬件使用超级页。这种做法适用于内存充足的情况，但在内存较小时，使用较小的页以更细的粒度进行分配和换出会更有效率。

5. 对实际内存配置的假设：

- xv6 假设系统有240MB的内存，而没有实际检测内存配置。虽然在 x86 上有几种方法可以检测实际的内存布局，但这些方法在 xv6 中没有实现。

6. 简单的内存分配：

- xv6 使用固定大小的4096字节页进行内存分配。这种简单的分配方式效率较高，但不如现代内存分配器灵活。现代操作系统通常会处理不同大小的内存分配请求，能够更有效地利用内存资源。

7. 与现代操作系统的对比：

- xv6 的内存管理和内存分配机制相对简单，适合教学和理解操作系统的基本概念。相比之下，现代操作系统的内存管理更为复杂，能够处理更多种类的内存请求，并在内存利用率和分配效率之间取得平衡。

源码解读

usys.S

这是 `usys.S` 文件的一部分代码，这段代码是由 `usys.pl` 脚本自动生成的，主要功能是为用户空间的系统调用提供汇编代码的包装。下面是对这段代码的详尽注释和解释：

```
1  # 由 usys.pl 自动生成 - 请勿编辑
2  #include "kernel/syscall.h" # 包含系统调用号的定义
3
4  # 定义全局符号 fork，使其在其他文件中可用
5  .global fork
6  fork:
7      # 将系统调用号 SYS_fork 加载到寄存器 a7 中
8      li a7, SYS_fork
9      # 触发系统调用，进入内核模式
10     ecall
11     # 返回用户模式
12     ret
13
14 # 定义全局符号 exit，使其在其他文件中可用
15 .global exit
16 exit:
17     li a7, SYS_exit
18     ecall
19     ret
20
```

```
21 # 定义全局符号 wait, 使其在其他文件中可用
22 .global wait
23 wait:
24     li a7, SYS_wait
25     ecall
26     ret
27
28 # 定义全局符号 pipe, 使其在其他文件中可用
29 .global pipe
30 pipe:
31     li a7, SYS_pipe
32     ecall
33     ret
34
35 # 定义全局符号 read, 使其在其他文件中可用
36 .global read
37 read:
38     li a7, SYS_read
39     ecall
40     ret
41
42 # 定义全局符号 write, 使其在其他文件中可用
43 .global write
44 write:
45     li a7, SYS_write
46     ecall
47     ret
48
49 # 定义全局符号 close, 使其在其他文件中可用
50 .global close
51 close:
52     li a7, SYS_close
53     ecall
54     ret
55
56 # 定义全局符号 kill, 使其在其他文件中可用
57 .global kill
58 kill:
59     li a7, SYS_kill
60     ecall
61     ret
62
```

```
63 # 定义全局符号 exec, 使其在其他文件中可用
64 .global exec
65 exec:
66     li a7, SYS_exec
67     ecall
68     ret
69
70 # 定义全局符号 open, 使其在其他文件中可用
71 .global open
72 open:
73     li a7, SYS_open
74     ecall
75     ret
76
77 # 定义全局符号 mknod, 使其在其他文件中可用
78 .global mknod
79 mknod:
80     li a7, SYS_mknod
81     ecall
82     ret
83
84 # 定义全局符号 unlink, 使其在其他文件中可用
85 .global unlink
86 unlink:
87     li a7, SYS_unlink
88     ecall
89     ret
90
91 # 定义全局符号 fstat, 使其在其他文件中可用
92 .global fstat
93 fstat:
94     li a7, SYS_fstat
95     ecall
96     ret
97
98 # 定义全局符号 link, 使其在其他文件中可用
99 .global link
100 link:
101     li a7, SYS_link
102     ecall
103     ret
104
```

```
105 # 定义全局符号 mkdir，使其在其他文件中可用
106 .global mkdir
107 mkdir:
108     li a7, SYS_mkdir
109     ecall
110     ret
111
112 # 定义全局符号 chdir，使其在其他文件中可用
113 .global chdir
114 chdir:
115     li a7, SYS_chdir
116     ecall
117     ret
118
119 # 定义全局符号 dup，使其在其他文件中可用
120 .global dup
121 dup:
122     li a7, SYS_dup
123     ecall
124     ret
125
126 # 定义全局符号 getpid，使其在其他文件中可用
127 .global getpid
128 getpid:
129     li a7, SYS_getpid
130     ecall
131     ret
132
133 # 定义全局符号 sbrk，使其在其他文件中可用
134 .global sbrk
135 sbrk:
136     li a7, SYS_sbrk
137     ecall
138     ret
139
140 # 定义全局符号 sleep，使其在其他文件中可用
141 .global sleep
142 sleep:
143     li a7, SYS_sleep
144     ecall
145     ret
146
```

```

147 # 定义全局符号 uptime, 使其在其他文件中可用
148 .global uptime
149 uptime:
150     li a7, SYS_uptime
151     ecall
152     ret

```

代码作用

这段代码的主要作用是为每一个系统调用提供汇编级的包装。每个系统调用的包装基本结构如下：

1. `li a7, SYS_<syscall_name>`: 将系统调用号加载到寄存器 `a7` 中。
2. `ecall`: 触发系统调用指令，切换到内核模式。
3. `ret`: 从系统调用返回到用户模式。

系统调用的过程

1. 调用接口: 用户程序调用一个系统调用，比如 `fork`。
2. 汇编包装: 对应的汇编包装代码执行，将系统调用号加载到 `a7` 寄存器中，然后通过 `ecall` 指令将控制权交给内核。
3. 内核处理: 内核根据 `a7` 中的系统调用号执行相应的内核功能（如进程创建、文件操作等）。
4. 返回结果: 内核完成操作后，通过 `ret` 指令返回用户空间。

系统调用的意义

系统调用是用户程序与操作系统内核交互的唯一接口。通过这些系统调用，用户程序可以执行诸如创建进程、打开文件、读写文件等操作，而无需直接操作硬件或内核数据结构，从而保证了系统的安全性和稳定性。

syscall.c

下面是对这段代码的详尽注释：

```

1 #include "types.h"

```

```

2  #include "param.h"
3  #include "memlayout.h"
4  #include "riscv.h"
5  #include "spinlock.h"
6  #include "proc.h"
7  #include "syscall.h"
8  #include "defs.h"
9
10 // 从当前进程中获取指定地址 addr 处的 uint64 值，并将其存储在 ip 中。
11 int
12 fetchaddr(uint64 addr, uint64 *ip)
13 {
14     struct proc *p = myproc(); // 获取当前进程指针
15     if(addr >= p->sz || addr+sizeof(uint64) > p->sz) // 检查地址是
        否超出进程内存范围，防止越界
16         return -1;
17     if(copyin(p->pagetable, (char *)ip, addr, sizeof(*ip)) != 0)
        // 从用户空间复制数据到内核空间
18         return -1;
19     return 0;
20 }
21
22 // 从当前进程中获取指定地址 addr 处的以空字符 '\0' 结尾的字符串，并将其存
    储在 buf 中。
23 // 返回字符串的长度（不包括 '\0'），如果发生错误，返回 -1。
24 int
25 fetchstr(uint64 addr, char *buf, int max)
26 {
27     struct proc *p = myproc(); // 获取当前进程指针
28     if(copyinstr(p->pagetable, buf, addr, max) < 0) // 从用户空间复
        制字符串到内核空间
29         return -1;
30     return strlen(buf); // 返回字符串长度
31 }
32
33 // 获取第 n 个系统调用参数的原始值（不做合法性检查）。
34 static uint64
35 argraw(int n)
36 {
37     struct proc *p = myproc(); // 获取当前进程指针
38     switch (n) { // 根据 n 的值返回对应寄存器中的值
39     case 0:

```

```

40     return p->trapframe->a0;
41 case 1:
42     return p->trapframe->a1;
43 case 2:
44     return p->trapframe->a2;
45 case 3:
46     return p->trapframe->a3;
47 case 4:
48     return p->trapframe->a4;
49 case 5:
50     return p->trapframe->a5;
51 }
52 panic("argraw"); // 如果 n 的值不在 0 到 5 之间, 说明出现了意外情况
53 return -1;
54 }
55
56 // 获取第 n 个 32 位的系统调用参数, 并将其存储在 ip 中。
57 void
58 argint(int n, int *ip)
59 {
60     *ip = argraw(n); // 调用 argraw 获取参数值并存储在 ip 中
61 }
62
63 // 获取第 n 个系统调用参数作为指针, 并存储在 ip 中。
64 void
65 argaddr(int n, uint64 *ip)
66 {
67     *ip = argraw(n); // 调用 argraw 获取参数值并存储在 ip 中
68 }
69
70 // 获取第 n 个系统调用参数作为字符串, 复制到 buf 中, 最多复制 max 字节。
71 // 返回字符串的长度 (包括 '\0'), 如果发生错误, 返回 -1。
72 int
73 argstr(int n, char *buf, int max)
74 {
75     uint64 addr;
76     argaddr(n, &addr); // 获取第 n 个参数作为地址
77     return fetchstr(addr, buf, max); // 获取字符串并返回其长度
78 }
79
80 // 声明系统调用处理函数的原型, 这些函数会在系统调用时被执行。
81 extern uint64 sys_fork(void);

```

```
82 extern uint64 sys_exit(void);
83 extern uint64 sys_wait(void);
84 extern uint64 sys_pipe(void);
85 extern uint64 sys_read(void);
86 extern uint64 sys_kill(void);
87 extern uint64 sys_exec(void);
88 extern uint64 sys_fstat(void);
89 extern uint64 sys_chdir(void);
90 extern uint64 sys_dup(void);
91 extern uint64 sys_getpid(void);
92 extern uint64 sys_sbrk(void);
93 extern uint64 sys_sleep(void);
94 extern uint64 sys_uptime(void);
95 extern uint64 sys_open(void);
96 extern uint64 sys_write(void);
97 extern uint64 sys_mknod(void);
98 extern uint64 sys_unlink(void);
99 extern uint64 sys_link(void);
100 extern uint64 sys_mkdir(void);
101 extern uint64 sys_close(void);
102
103 // 一个数组，将系统调用号映射到对应的系统调用处理函数。
104 static uint64 (*syscalls[])(void) = {
105     [SYS_fork]    sys_fork,
106     [SYS_exit]    sys_exit,
107     [SYS_wait]    sys_wait,
108     [SYS_pipe]    sys_pipe,
109     [SYS_read]    sys_read,
110     [SYS_kill]    sys_kill,
111     [SYS_exec]    sys_exec,
112     [SYS_fstat]   sys_fstat,
113     [SYS_chdir]   sys_chdir,
114     [SYS_dup]     sys_dup,
115     [SYS_getpid]  sys_getpid,
116     [SYS_sbrk]    sys_sbrk,
117     [SYS_sleep]   sys_sleep,
118     [SYS_uptime]  sys_uptime,
119     [SYS_open]    sys_open,
120     [SYS_write]   sys_write,
121     [SYS_mknod]   sys_mknod,
122     [SYS_unlink]  sys_unlink,
123     [SYS_link]    sys_link,
```



```

124 [SYS_mkdir]    sys_mkdir,
125 [SYS_close]   sys_close,
126 };
127
128 // 系统调用处理函数，处理所有系统调用。
129 void
130 syscall(void)
131 {
132     int num;
133     struct proc *p = myproc(); // 获取当前进程指针
134
135     num = p->trapframe->a7; // 获取系统调用号
136     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) { // 检
        查系统调用号是否有效
137         // 调用对应的系统调用处理函数，并将返回值存储在 a0 中
138         p->trapframe->a0 = syscalls[num]();
139     } else {
140         // 如果系统调用号无效，打印错误信息并返回 -1
141         printf("%d %s: unknown sys call %d\n",
142             p->pid, p->name, num);
143         p->trapframe->a0 = -1;
144     }
145 }

```

代码的主要功能和调用点：

- **fetchaddr** 和 **fetchstr**：用于从用户进程的地址空间中获取数据（一个整数或一个字符串），并将其拷贝到内核空间。这些函数主要用于处理系统调用参数。
- **argraw**, **argint**, **argaddr**, **argstr**：用于提取系统调用的参数。系统调用的参数在用户进程的寄存器中（a0-a5），这些函数将寄存器中的值提取出来并进行类型转换。
- **syscall**：是处理所有系统调用的主函数。根据系统调用号，从 **syscalls** 数组中查找对应的处理函数，并调用它来执行实际的系统调用操作。

代码的重要性：

这段代码是操作系统处理系统调用的关键部分。系统调用是用户程序和操作系统内核交互的唯一途径，所有的文件操作、进程管理、内存管理等系统功能都是通过系统调用实现的。这段代码的作用是将用户程序发出的系统调用请求转发给内核中对应的处理函数，从而实现这些操作。

proc.c

下面是对这段代码的详尽注释：

```
1  #include "types.h"
2  #include "param.h"
3  #include "memlayout.h"
4  #include "riscv.h"
5  #include "spinlock.h"
6  #include "proc.h"
7  #include "defs.h"
8
9  // 每个 CPU 的信息
10 struct cpu cpus[NCPU];
11
12 // 所有进程的全局数组
13 struct proc proc[NPROC];
14
15 // 初始进程的指针
16 struct proc *initproc;
17
18 int nextpid = 1; // 下一个要分配的进程 ID
19 struct spinlock pid_lock; // 保护进程 ID 分配的锁
20
21 extern void forkret(void); // fork 返回后执行的函数
22 static void freeproc(struct proc *p); // 释放进程资源的函数
23
24 extern char trampoline[]; // trampoline.S 中定义的 trampoline 地址
25
26 // 保护 wait 相关的锁，避免父进程在 wait() 时丢失 wakeup 信号。
27 // 必须在获取任何进程锁之前获取该锁。
28 struct spinlock wait_lock;
29
```

```

30 // 为每个进程的内核栈分配一页，并在高地址处映射它，
31 // 随后是一个无效的保护页。
32 void
33 proc_mapstacks(pagetable_t kpgtbl)
34 {
35     struct proc *p;
36
37     for(p = proc; p < &proc[NPROC]; p++) {
38         char *pa = kalloc(); // 分配一页内存作为内核栈
39         if(pa == 0)
40             panic("kalloc");
41         uint64 va = KSTACK((int) (p - proc)); // 计算内核栈的虚拟地址
42         kvmmap(kpgtbl, va, (uint64)pa, PGSIZE, PTE_R | PTE_W); //
映射内核栈
43     }
44 }
45
46 // 初始化进程表
47 void
48 procinit(void)
49 {
50     struct proc *p;
51
52     initlock(&pid_lock, "nextpid"); // 初始化 PID 分配锁
53     initlock(&wait_lock, "wait_lock"); // 初始化等待锁
54     for(p = proc; p < &proc[NPROC]; p++) {
55         initlock(&p->lock, "proc"); // 初始化每个进程的锁
56         p->state = UNUSED; // 设置进程状态为未使用
57         p->kstack = KSTACK((int) (p - proc)); // 设置内核栈的虚拟地
址
58     }
59 }
60
61 // 必须在禁用中断的情况下调用，以避免与进程被移动到其他 CPU 竞争。
62 int
63 cpuid()
64 {
65     int id = r_tp(); // 从线程寄存器读取当前 CPU ID
66     return id;
67 }
68
69 // 返回当前 CPU 的 cpu 结构体。

```

```
70 // 中断必须被禁用。
71 struct cpu*
72 mycpu(void)
73 {
74     int id = cpuid(); // 获取当前 CPU ID
75     struct cpu *c = &cpus[id]; // 返回当前 CPU 的 cpu 结构体
76     return c;
77 }
78
79 // 返回当前进程的结构体指针，如果没有进程运行，则返回 0。
80 struct proc*
81 myproc(void)
82 {
83     push_off(); // 禁用中断
84     struct cpu *c = mycpu(); // 获取当前 CPU 的结构体
85     struct proc *p = c->proc; // 获取当前 CPU 正在运行的进程
86     pop_off(); // 启用中断
87     return p;
88 }
89
90 // 分配一个新的进程 ID
91 int
92 allocpid()
93 {
94     int pid;
95
96     acquire(&pid_lock); // 获取 PID 分配锁
97     pid = nextpid; // 获取当前的 PID
98     nextpid = nextpid + 1; // 增加下一个 PID
99     release(&pid_lock); // 释放锁
100
101     return pid;
102 }
103
104 // 在进程表中查找一个未使用的进程结构体。
105 // 如果找到，则初始化并返回，锁持有 p->lock。
106 // 如果没有空闲的进程或内存分配失败，则返回 0。
107 static struct proc*
108 allocproc(void)
109 {
110     struct proc *p;
111
```

```

112     for(p = proc; p < &proc[NPROC]; p++) {
113         acquire(&p->lock); // 获取进程的锁
114         if(p->state == UNUSED) {
115             goto found; // 找到一个未使用的进程
116         } else {
117             release(&p->lock); // 释放锁
118         }
119     }
120     return 0;
121
122 found:
123     p->pid = allocpid(); // 分配一个新的进程 ID
124     p->state = USED; // 设置进程状态为已使用
125
126     // 分配一个陷阱帧页面
127     if((p->trapframe = (struct trapframe *)kalloc()) == 0){
128         freeproc(p); // 释放进程资源
129         release(&p->lock); // 释放锁
130         return 0;
131     }
132
133     // 创建一个空的用户页表
134     p->pagetable = proc_pagetable(p);
135     if(p->pagetable == 0){
136         freeproc(p); // 释放进程资源
137         release(&p->lock); // 释放锁
138         return 0;
139     }
140
141     // 设置新的上下文以便在 forkret 处开始执行，并返回用户空间。
142     memset(&p->context, 0, sizeof(p->context));
143     p->context.ra = (uint64)forkret;
144     p->context.sp = p->kstack + PGSIZE; // 设置内核栈的栈顶指针
145
146     return p;
147 }
148
149 // 释放进程结构体及其相关资源，包括用户页。
150 // 必须持有 p->lock。
151 static void
152 freeproc(struct proc *p)
153 {

```

```

154     if(p->trapframe)
155         kfree((void*)p->trapframe); // 释放陷阱帧内存
156     p->trapframe = 0;
157     if(p->pagetable)
158         proc_freepagetable(p->pagetable, p->sz); // 释放页表和用户内
存
159     p->pagetable = 0;
160     p->sz = 0;
161     p->pid = 0;
162     p->parent = 0;
163     p->name[0] = 0;
164     p->chan = 0;
165     p->killed = 0;
166     p->xstate = 0;
167     p->state = UNUSED; // 设置进程状态为未使用
168 }
169
170 // 为给定进程创建一个用户页表，没有用户内存，但包含 trampoline 和
trapframe 页面。
171 pagetable_t
172 proc_pagetable(struct proc *p)
173 {
174     pagetable_t pagetable;
175
176     // 创建一个空的页表
177     pagetable = uvmcreate();
178     if(pagetable == 0)
179         return 0;
180
181     // 映射 trampoline 代码（用于系统调用返回），仅供内核使用。
182     if(mappages(pagetable, TRAMPOLINE, PGSIZE,
183                 (uint64)trampoline, PTE_R | PTE_X) < 0){
184         uvmfree(pagetable, 0); // 释放页表
185         return 0;
186     }
187
188     // 映射 trapframe 页面（trampoline.S 使用）。
189     if(mappages(pagetable, TRAPFRAME, PGSIZE,
190                 (uint64)(p->trapframe), PTE_R | PTE_W) < 0){
191         uvmunmap(pagetable, TRAMPOLINE, 1, 0); // 取消映射
trampoline
192         uvmfree(pagetable, 0); // 释放页表

```

```

193     return 0;
194 }
195
196     return pagetable;
197 }
198
199 // 释放进程的页表，并释放它引用的物理内存。
200 void
201 proc_freepagetable(pagetable_t pagetable, uint64 sz)
202 {
203     uvmunmap(pagetable, TRAMPOLINE, 1, 0); // 取消映射 trampoline
204     uvmunmap(pagetable, TRAPFRAME, 1, 0); // 取消映射 trapframe
205     uvmfree(pagetable, sz); // 释放页表
206 }
207
208 // 这是一个用户程序，它调用 exec("/init")。
209 // 该程序是由 ../user/initcode.S 生成的。
210 uchar initcode[] = {
211     0x17, 0x05, 0x00, 0x00, 0x13, 0x05, 0x45, 0x02,
212     0x97, 0x05, 0x00, 0x00, 0x93, 0x85, 0x35, 0x02,
213     0x93, 0x08, 0x70, 0x00, 0x73, 0x00, 0x00, 0x00,
214     0x93, 0x08, 0x20, 0x00, 0x73, 0x00, 0x00, 0x00,
215     0xef, 0xf0, 0x9f, 0xff, 0x2f, 0x69, 0x6e, 0x69,
216     0x74, 0x00
217
218     , 0x00, 0x24, 0x00, 0x00, 0x00, 0x00,
219     0x00, 0x00, 0x00, 0x00
220 };
221
222 // 设置第一个用户进程
223 void
224 userinit(void)
225 {
226     struct proc *p;
227
228     p = allocproc(); // 分配并初始化第一个进程
229     initproc = p;
230
231     // 分配一个用户页面，并将 initcode 的指令和数据拷贝到该页面中。
232     uvmfirst(p->pagetable, initcode, sizeof(initcode));
233     p->sz = PGSIZE; // 设置进程的大小为一页
234

```

```

235 // 准备第一次从内核返回用户态
236 p->trapframe->epc = 0; // 用户程序计数器
237 p->trapframe->sp = PGSIZE; // 用户栈指针
238
239 safestrcpy(p->name, "initcode", sizeof(p->name));
240 p->cwd = namei("/"); // 设置当前工作目录
241
242 p->state = RUNNABLE; // 设置进程状态为可运行
243
244 release(&p->lock); // 释放进程锁
245 }
246
247 // 增加或减少用户内存 n 字节。
248 // 成功返回 0，失败返回 -1。
249 int
250 growproc(int n)
251 {
252     uint64 sz;
253     struct proc *p = myproc();
254
255     sz = p->sz; // 获取当前进程的大小
256     if(n > 0){
257         if((sz = uvmalloc(p->pagetable, sz, sz + n, PTE_W)) == 0) {
258             return -1; // 内存分配失败
259         }
260     } else if(n < 0){
261         sz = uvmdealloc(p->pagetable, sz, sz + n); // 释放内存
262     }
263     p->sz = sz; // 更新进程的大小
264     return 0;
265 }
266
267 // 创建一个新进程，复制父进程。
268 // 设置子进程的内存栈，以便从 fork() 系统调用返回。
269 int
270 fork(void)
271 {
272     int i, pid;
273     struct proc *np;
274     struct proc *p = myproc();
275
276     // 分配进程

```



```

277     if((np = allocproc()) == 0){
278         return -1;
279     }
280
281     // 将父进程的用户内存复制到子进程
282     if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
283         freeproc(np); // 释放进程资源
284         release(&np->lock); // 释放锁
285         return -1;
286     }
287     np->sz = p->sz; // 设置子进程的大小
288
289     // 复制用户寄存器
290     *(np->trapframe) = *(p->trapframe);
291
292     // 在子进程中设置 fork 返回值为 0
293     np->trapframe->a0 = 0;
294
295     // 增加打开文件描述符的引用计数
296     for(i = 0; i < NOFILE; i++)
297         if(p->ofile[i])
298             np->ofile[i] = filedup(p->ofile[i]);
299     np->cwd = idup(p->cwd); // 复制当前工作目录
300
301     safestrcpy(np->name, p->name, sizeof(p->name)); // 复制进程名
302     称
303
304     pid = np->pid; // 获取子进程的 PID
305
306     release(&np->lock); // 释放子进程的锁
307
308     acquire(&wait_lock);
309     np->parent = p; // 设置子进程的父进程
310     release(&wait_lock);
311
312     acquire(&np->lock);
313     np->state = RUNNABLE; // 设置子进程状态为可运行
314     release(&np->lock);
315
316     return pid;
317 }

```

```

318 // 将 p 的孤儿子进程交给 init 进程处理。
319 // 调用者必须持有 wait_lock。
320 void
321 reparent(struct proc *p)
322 {
323     struct proc *pp;
324
325     for(pp = proc; pp < &proc[NPROC]; pp++){
326         if(pp->parent == p){
327             pp->parent = initproc; // 将子进程的父进程设置为 init
328             wakeup(initproc); // 唤醒 init 进程
329         }
330     }
331 }
332
333 // 退出当前进程，不返回。
334 // 退出的进程将保持在 ZOMBIE 状态，直到其父进程调用 wait()。
335 void
336 exit(int status)
337 {
338     struct proc *p = myproc();
339
340     if(p == initproc)
341         panic("init exiting");
342
343     // 关闭所有打开的文件
344     for(int fd = 0; fd < NOFILE; fd++){
345         if(p->ofile[fd]){
346             struct file *f = p->ofile[fd];
347             fileclose(f); // 关闭文件
348             p->ofile[fd] = 0;
349         }
350     }
351
352     begin_op();
353     iput(p->cwd); // 释放当前工作目录
354     end_op();
355     p->cwd = 0;
356
357     acquire(&wait_lock);
358
359     // 将所有子进程交给 init 进程

```

```

360     reparent(p);
361
362     // 父进程可能在 wait() 中睡眠，唤醒它。
363     wakeup(p->parent);
364
365     acquire(&p->lock);
366
367     p->xstate = status; // 设置退出状态
368     p->state = ZOMBIE; // 设置进程状态为 ZOMBIE
369
370     release(&wait_lock);
371
372     // 进入调度器，永远不返回。
373     sched();
374     panic("zombie exit");
375 }
376
377 // 等待一个子进程退出并返回其 PID。
378 // 如果该进程没有子进程，返回 -1。
379 int
380 wait(uint64 addr)
381 {
382     struct proc *pp;
383     int havekids, pid;
384     struct proc *p = myproc();
385
386     acquire(&wait_lock);
387
388     for(;;){
389         // 扫描进程表，查找已退出的子进程。
390         havekids = 0;
391         for(pp = proc; pp < &proc[NPROC]; pp++){
392             if(pp->parent == p){
393                 // 确保子进程不在 exit() 或 swtch() 中。
394                 acquire(&pp->lock);
395
396                 havekids = 1;
397                 if(pp->state == ZOMBIE){
398                     // 找到了一个已退出的子进程。
399                     pid = pp->pid;
400                     if(addr != 0 && copyout(p->pagetable, addr, (char
*)&pp->xstate,

```

```

401         sizeof(pp->xstate)) < 0) {
402             release(&pp->lock);
403             release(&wait_lock);
404             return -1;
405         }
406         freeproc(pp); // 释放子进程资源
407         release(&pp->lock);
408         release(&wait_lock);
409         return pid;
410     }
411     release(&pp->lock);
412 }
413 }
414
415 // 如果没有子进程，则没有必要等待。
416 if(!havekids || killed(p)){
417     release(&wait_lock);
418     return -1;
419 }
420
421 // 等待一个子进程退出。
422 sleep(p, &wait_lock); // 在 wait_lock 上睡眠
423 }
424 }
425
426 // 每个 CPU 的进程调度器。
427 // 每个 CPU 设置完毕后调用 scheduler()。
428 // 调度器永远不返回。它循环执行：
429 // - 选择一个进程来运行。
430 // - 切换到该进程。
431 // - 最终该进程通过 swtch 返回调度器。
432 void
433 scheduler(void)
434 {
435     struct proc *p;
436     struct cpu *c = mycpu();
437
438     c->proc = 0;
439     for(;;){
440         // 最近运行的进程可能已关闭中断；启用它们，以避免死锁。
441         intr_on();
442

```

```

443     for(p = proc; p < &proc[NPROC]; p++) {
444         acquire(&p->lock);
445         if(p->state == RUNNABLE) {
446             // 切换到选中的进程。进程有责任释放锁并在返回前重新获取锁。
447             p->state = RUNNING;
448             c->proc = p;
449             swtch(&c->context, &p->context); // 切换上下文
450
451             // 进程暂时不再运行。
452             // 它应该在返回前更改其状态。
453             c->proc = 0;
454         }
455         release(&p->lock);
456     }
457 }
458 }
459
460 // 切换到调度器。必须仅持有 p->lock，并且已更改 proc->state。
461 // 保存和恢复 intena，因为 intena 是该内核线程的属性，而不是 CPU 的属性。
462 void
463 sched(void)
464 {
465     int intena;
466     struct proc *p = myproc();
467
468     if(!holding(&p->lock))
469         panic("sched p->lock");
470     if(mycpu()->noff != 1)
471         panic("sched locks");
472     if(p->state == RUNNING)
473         panic("sched running");
474     if(intr_get())
475         panic("sched interruptible");
476
477     intena = mycpu()->intena;
478     swtch(&p->context, &mycpu()->context); // 切换到调度器上下文
479     mycpu()->intena = intena;
480 }
481
482 // 放弃 CPU
483

```

```
484 , 一个调度周期。
485 void
486 yield(void)
487 {
488     struct proc *p = myproc();
489     acquire(&p->lock);
490     p->state = RUNNABLE; // 设置进程状态为可运行
491     sched(); // 切换到调度器
492     release(&p->lock);
493 }
494
495 // fork 子进程的第一次调度将切换到 forkret。
496 void
497 forkret(void)
498 {
499     static int first = 1;
500
501     // 调度器仍然持有 p->lock。
502     release(&myproc()->lock);
503
504     if (first) {
505         // 文件系统初始化必须在常规进程的上下文中运行（例如，它调用 sleep），
506         // 因此不能在 main() 中运行。
507         fsinit(ROOTDEV);
508
509         first = 0;
510         // 确保其他内核看到 first=0。
511         __sync_synchronize();
512     }
513
514     usertrapret(); // 返回用户空间
515 }
516
517 // 原子地释放锁并在 chan 上睡眠。
518 // 被唤醒时重新获取锁。
519 void
520 sleep(void *chan, struct spinlock *lk)
521 {
522     struct proc *p = myproc();
523
524     // 必须获取 p->lock 以更改 p->state, 然后调用 sched。
```

```

525     // 一旦持有 p->lock, 可以保证不会错过任何唤醒 (wakeup 锁定 p-
    >lock),
526     // 因此可以释放 lk。
527
528     acquire(&p->lock); // 获取进程锁
529     release(lk); // 释放原锁
530
531     // 进入睡眠状态。
532     p->chan = chan;
533     p->state = SLEEPING; // 设置进程状态为睡眠
534
535     sched(); // 调度其他进程
536
537     // 清理工作。
538     p->chan = 0;
539
540     // 重新获取原始锁。
541     release(&p->lock);
542     acquire(lk);
543 }
544
545 // 唤醒在 chan 上睡眠的所有进程。
546 // 必须在没有持有任何 p->lock 的情况下调用。
547 void
548 wakeup(void *chan)
549 {
550     struct proc *p;
551
552     for(p = proc; p < &proc[NPROC]; p++) {
553         if(p != myproc()){
554             acquire(&p->lock);
555             if(p->state == SLEEPING && p->chan == chan) {
556                 p->state = RUNNABLE; // 将进程状态设置为可运行
557             }
558             release(&p->lock);
559         }
560     }
561 }
562
563 // 终止给定 pid 的进程。
564 // 受害者在尝试返回用户空间之前不会退出 (见 trap.c 中的 usertrap())。
565 int

```

```

566 kill(int pid)
567 {
568     struct proc *p;
569
570     for(p = proc; p < &proc[NPROC]; p++){
571         acquire(&p->lock);
572         if(p->pid == pid){
573             p->killed = 1; // 设置进程的杀死标志
574             if(p->state == SLEEPING){
575                 // 唤醒睡眠中的进程。
576                 p->state = RUNNABLE;
577             }
578             release(&p->lock);
579             return 0;
580         }
581         release(&p->lock);
582     }
583     return -1;
584 }
585
586 // 设置进程的杀死标志
587 void
588 setkilled(struct proc *p)
589 {
590     acquire(&p->lock);
591     p->killed = 1;
592     release(&p->lock);
593 }
594
595 // 检查进程是否被标记为杀死
596 int
597 killed(struct proc *p)
598 {
599     int k;
600
601     acquire(&p->lock);
602     k = p->killed;
603     release(&p->lock);
604     return k;
605 }
606
607 // 复制到用户地址或内核地址，

```



```

608 // 取决于 usr_dst 标志。
609 // 成功返回 0，错误返回 -1。
610 int
611 either_copyout(int user_dst, uint64 dst, void *src, uint64 len)
612 {
613     struct proc *p = myproc();
614     if(user_dst){
615         return copyout(p->pagetable, dst, src, len); // 复制到用户地
        址
616     } else {
617         memmove((char *)dst, src, len); // 复制到内核地址
618         return 0;
619     }
620 }
621
622 // 从用户地址或内核地址复制，
623 // 取决于 usr_src 标志。
624 // 成功返回 0，错误返回 -1。
625 int
626 either_copyin(void *dst, int user_src, uint64 src, uint64 len)
627 {
628     struct proc *p = myproc();
629     if(user_src){
630         return copyin(p->pagetable, dst, src, len); // 从用户地址复制
631     } else {
632         memmove(dst, (char*)src, len); // 从内核地址复制
633         return 0;
634     }
635 }
636
637 // 打印进程列表到控制台。用于调试。
638 // 当用户在控制台上键入 ^P 时运行。
639 // 不使用锁，以避免进一步卡住已经被卡住的机器。
640 void
641 procdump(void)
642 {
643     static char *states[] = {
644         [UNUSED]    "unused", // 未使用
645         [USED]      "used",    // 已使用
646         [SLEEPING]  "sleep ",  // 睡眠
647         [RUNNABLE]  "runble",   // 可运行
648         [RUNNING]   "run  ",   // 运行中

```

```

649     [ZOMBIE]     "zombie"    // 僵尸
650 };
651 struct proc *p;
652 char *state;
653
654 printf("\n");
655 for(p = proc; p < &proc[NPROC]; p++){
656     if(p->state == UNUSED)
657         continue;
658     if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
659         state = states[p->state]; // 设置状态字符串
660     else
661         state = "???";
662     printf("%d %s %s", p->pid, state, p->name); // 打印进程 ID、
        状态和名称
663     printf("\n");
664 }
665 }

```

这段代码实现了 xv6 操作系统中的进程管理模块，包括进程的创建、调度、退出、等待、内存管理等功能。主要流程包括：

1. 初始化进程表和 CPU 结构体。
2. 为每个进程分配内核栈和页表。
3. 实现了进程的创建（`fork`）、退出（`exit`）和等待子进程（`wait`）。
4. 通过调度器（`scheduler`）管理进程的执行。
5. 实现了进程间的唤醒和睡眠机制。

实验内容

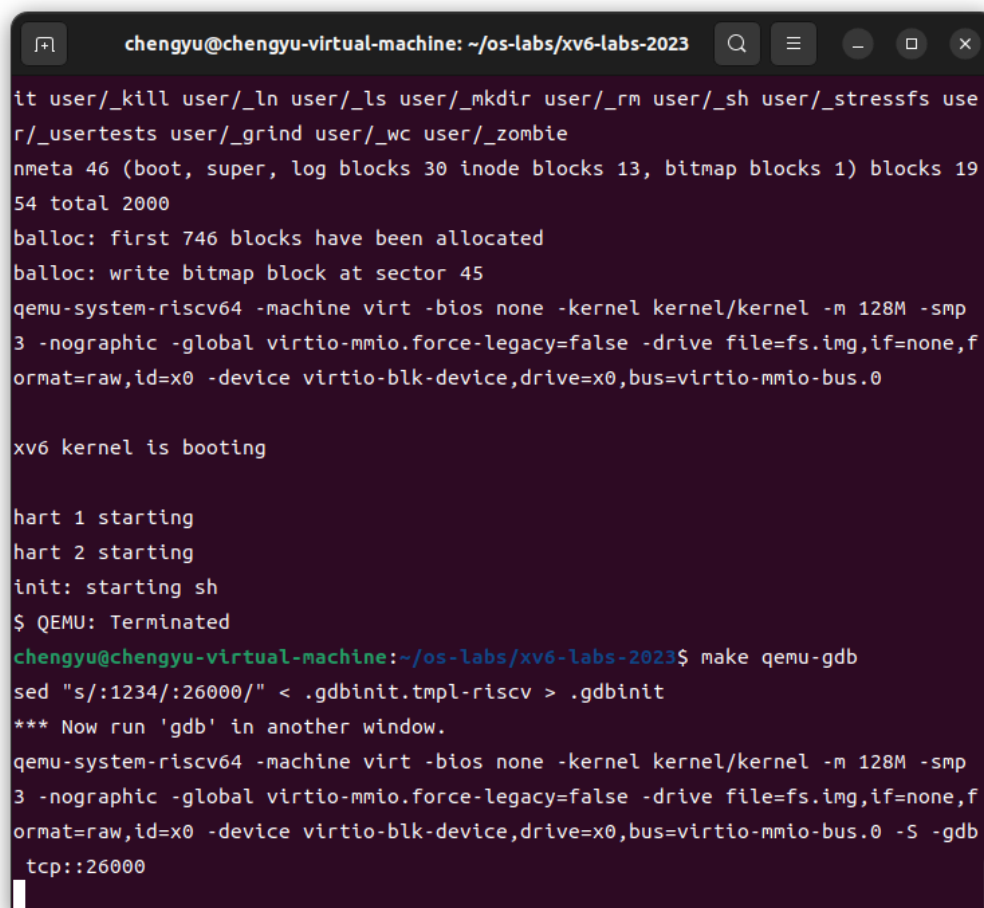
1. Using gdb(easy)

实验目的

- 学习如何使用 GDB 进行调试本 xv6。

实验步骤

- 在终端输入：`make qemu-gdb`。然后再打开一个终端，运行 `gdb-multiarch -x .gdbinit`。这将运行 `.gdbinit` 中的命令，也就是开启远程调试功能，并设置 `arch` 架构为 `riscv64`。

A terminal window titled 'chengyu@chengyu-virtual-machine: ~/os-labs/xv6-labs-2023'. The terminal shows the output of a previous run, including file operations, block allocation, and the start of the xv6 kernel. It then shows the command 'make qemu-gdb' being executed, which generates a .gdbinit file and prints instructions to run 'gdb' in another window. The terminal ends with the command 'gdb tcp::26000' entered.

```
chengyu@chengyu-virtual-machine: ~/os-labs/xv6-labs-2023
it user/_kill user/_ln user/_ls user/_mkdir user/_rm user/_sh user/_stressfs use
r/_usertests user/_grind user/_wc user/_zombie
nmeta 46 (boot, super, log blocks 30 inode blocks 13, bitmap blocks 1) blocks 19
54 total 2000
balloc: first 746 blocks have been allocated
balloc: write bitmap block at sector 45
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp
3 -nographic -global virtio-mmio.force-legacy=false -drive file=fs.img,if=none,f
ormat=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ QEMU: Terminated
chengyu@chengyu-virtual-machine:~/os-labs/xv6-labs-2023$ make qemu-gdb
sed "s/:1234/:26000/" < .gdbinit.tmpl-riscv > .gdbinit
*** Now run 'gdb' in another window.
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp
3 -nographic -global virtio-mmio.force-legacy=false -drive file=fs.img,if=none,f
ormat=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0 -S -gdb
tcp::26000
```

- 在GDB中运行以下指令：

```

1 (gdb) b syscall
2 Breakpoint 1 at 0x80001fe0: file kernel/syscall.c, line
  133.
3 (gdb) c
4 Continuing.
5 [Switching to Thread 1.3]
6
7 Thread 3 hit Breakpoint 1, syscall () at
  kernel/syscall.c:133
8 133      {
9 (gdb) layout src
10 (gdb) backtrace

```

`b syscall` 将在函数 `syscall` 处设置断点；`c` 将会运行到此断点时等待调试指令；`layout src` 将会开启一个窗口展示调试时的源代码；`backtrace` 将会打印堆栈回溯（stack backtrace）。

The screenshot shows a GDB terminal window with the following content:

```

chengyu@chengyu-virtual-machine: ~/os-labs/xv6-labs-2023
kernel/syscall.c
B+> 133 {
      134     int num;
      135     struct proc *p = myproc();
      136
      137     num = p->trapframe->a7;
      138     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
      139         // Use num to lookup the system call function for num, call it,
      140         // and store its return value in p->trapframe->a0
      141         p->trapframe->a0 = syscalls[num]();
      142     } else {
      143         printf("%d %s: unknown sys call %d\n",
      144             p->pid, p->name, num);
      145         p->trapframe->a0 = -1;
remote Thread 1.1 In: syscall                                L133 PC: 0x8000203e
(gdb) backtrace
#0  syscall () at kernel/syscall.c:133
#1  0x0000000080001d72 in usertrap () at kernel/trap.c:67
#2  0x0505050505050505 in ?? ()
(gdb)

```

- Q1: Looking at the backtrace output, which function called `syscall`?
- A: 通过堆栈回溯可以看到，函数 `usertrap()` 调用了 `syscall()` 函数。
- 输入几个 `n` 命令，使其执行 `struct proc *p = myproc();` 并打印 `*p` 的值，它是一个 `proc` 结构体。

- 1 (gdb) n
- 2 (gdb) n
- 3 (gdb) p/x *p

```

kernel/syscall.c
133 {
134     int num;
135     struct proc *p = myproc();
136
137     num = p->trapframe->a7;
138     if (num > 0 && num < NELEM(syscalls) && syscalls[num]) {
139         // Use num to lookup the system call function for num, call it,
140         // and store its return value in p->trapframe->a0
141         p->trapframe->a0 = syscalls[num]();
142     } else {
143         printf("ud %s: unknown sys call %d\n",
144             p->pid, p->name, num);
145         p->trapframe->a0 = -1;
146     }
147 }
148
149
150
151
152
153
154
155
156
157
158

remote Thread 1.1 In: syscall
L137 PC: 0x80002854
s1 = {lock = {locked = 0x0, name = 0x80001b8, cpu = 0x0}, state = 0x4,
chan = 0x0, killed = 0x0, xstate = 0x0, pid = 0x1, parent = 0x0,
kstack = 0x3fffffd000, sz = 0x1000, pagetable = 0x87773000,
trapframe = 0x87774000, context = {ra = 0x800014c2, sp = 0x3fffffd70,
s0 = 0x3fffffdea0, s1 = 0x8000bd70, s2 = 0x80008940, s3 = 0x1,
s4 = 0x3fffffde0, s5 = 0x8000ebf8, s6 = 0x3, s7 = 0x80019a10, s8 = 0x1,
s9 = 0x80019b38, s10 = 0x4, s11 = 0x0}, ofile = {0x0 <repeats 16 times>},
--Type <RET> for more, q to quit, c to continue without paging--

```

```

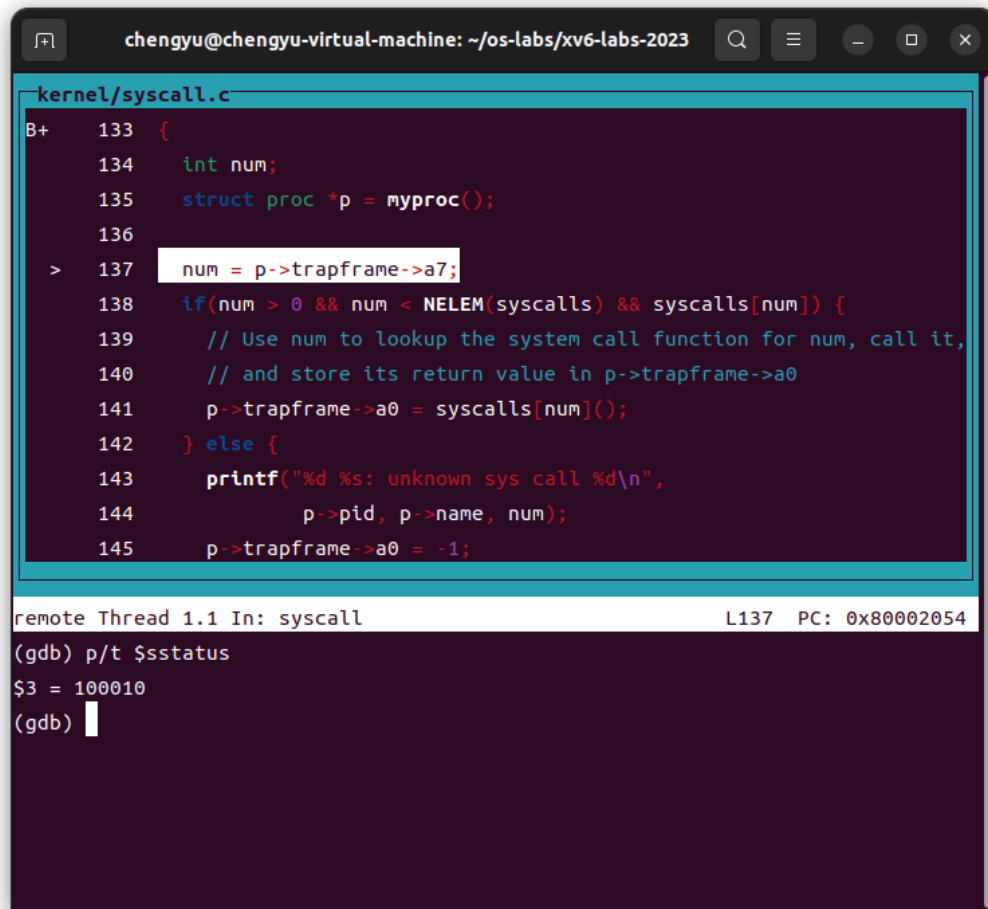
kernel/syscall.c
133 {
134     int num;
135     struct proc *p = myproc();
136
137     num = p->trapframe->a7;
138     if (num > 0 && num < NELEM(syscalls) && syscalls[num]) {
139         // Use num to lookup the system call function for num, call it,
140         // and store its return value in p->trapframe->a0
141         p->trapframe->a0 = syscalls[num]();
142     } else {
143         printf("ud %s: unknown sys call %d\n",
144             p->pid, p->name, num);
145         p->trapframe->a0 = -1;
146     }
147 }
148
149
150
151
152
153
154
155
156
157
158

remote Thread 1.1 In: syscall
L137 PC: 0x80002854
s1 = {lock = {locked = 0x0, name = 0x80001b8, cpu = 0x0}, state = 0x4,
chan = 0x0, killed = 0x0, xstate = 0x0, pid = 0x1, parent = 0x0,
kstack = 0x3fffffd000, sz = 0x1000, pagetable = 0x87773000,
trapframe = 0x87774000, context = {ra = 0x800014c2, sp = 0x3fffffd70,
s0 = 0x3fffffdea0, s1 = 0x8000bd70, s2 = 0x80008940, s3 = 0x1,
s4 = 0x3fffffde0, s5 = 0x8000ebf8, s6 = 0x3, s7 = 0x80019a10, s8 = 0x1,
s9 = 0x80019b38, s10 = 0x4, s11 = 0x0}, ofile = {0x0 <repeats 16 times>},
--Type <RET> for more, q to quit, c to continue without paging--c d = 0x8001ce80, name = {0x69, 0x6e, 0x69, 0x74, 0x63, 0x6f, 0x64, 0x65, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}}
(gdb) p p->trapframe->a7
s2 = 7
(gdb)

```

- Q2: What is the value of `p->trapframe->a7` and what does that value represent? (Hint: look `user/initcode.S`, the first user program `xv6` starts.)
- A: 根据参考教材 `xv6 book` 第二章和 `user/initcode.S` 中的代码可知，这个 `a7` 寄存器中保存了将要执行的系统调用号。这里的系统调用号为 `7`，在 `kernel/syscall.h` 中可以找到，这个系统调用为 `SYS_exec`。
- 输入 GDB 命令来查看 `sstatus` 寄存器。通过 `p/t` 以二进制打印。

```
1 (gdb) p/t $sstatus
2 $4 = 100010
3 (gdb)
```



```
chengyu@chengyu-virtual-machine: ~/os-labs/xv6-labs-2023
kernel/syscall.c
B+ 133 {
    134     int num;
    135     struct proc *p = myproc();
    136
    > 137     num = p->trapframe->a7;
    138     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    139         // Use num to lookup the system call function for num, call it,
    140         // and store its return value in p->trapframe->a0
    141         p->trapframe->a0 = syscalls[num]();
    142     } else {
    143         printf("%d %s: unknown sys call %d\n",
    144             p->pid, p->name, num);
    145         p->trapframe->a0 = -1;
    }

remote Thread 1.1 In: syscall L137 PC: 0x80002054
(gdb) p/t $sstatus
$3 = 100010
(gdb)
```

分析

sstatus 寄存器的二进制值包含了多个标志位，这些标志位用于表示 CPU 的当前模式以及其他状态信息。以下是主要相关标志位的说明：

- 位 1 (**SPP**): Supervisor Previous Privilege，表示 CPU 在进入当前模式前所处的模式。**0** 表示用户模式（User Mode），**1** 表示监督模式（Supervisor Mode）。
- 位 5 (**SIE**): Supervisor Interrupt Enable，表示是否启用了监督模式下的中断。

解释

输出 **100010** 可以分解为以下位：

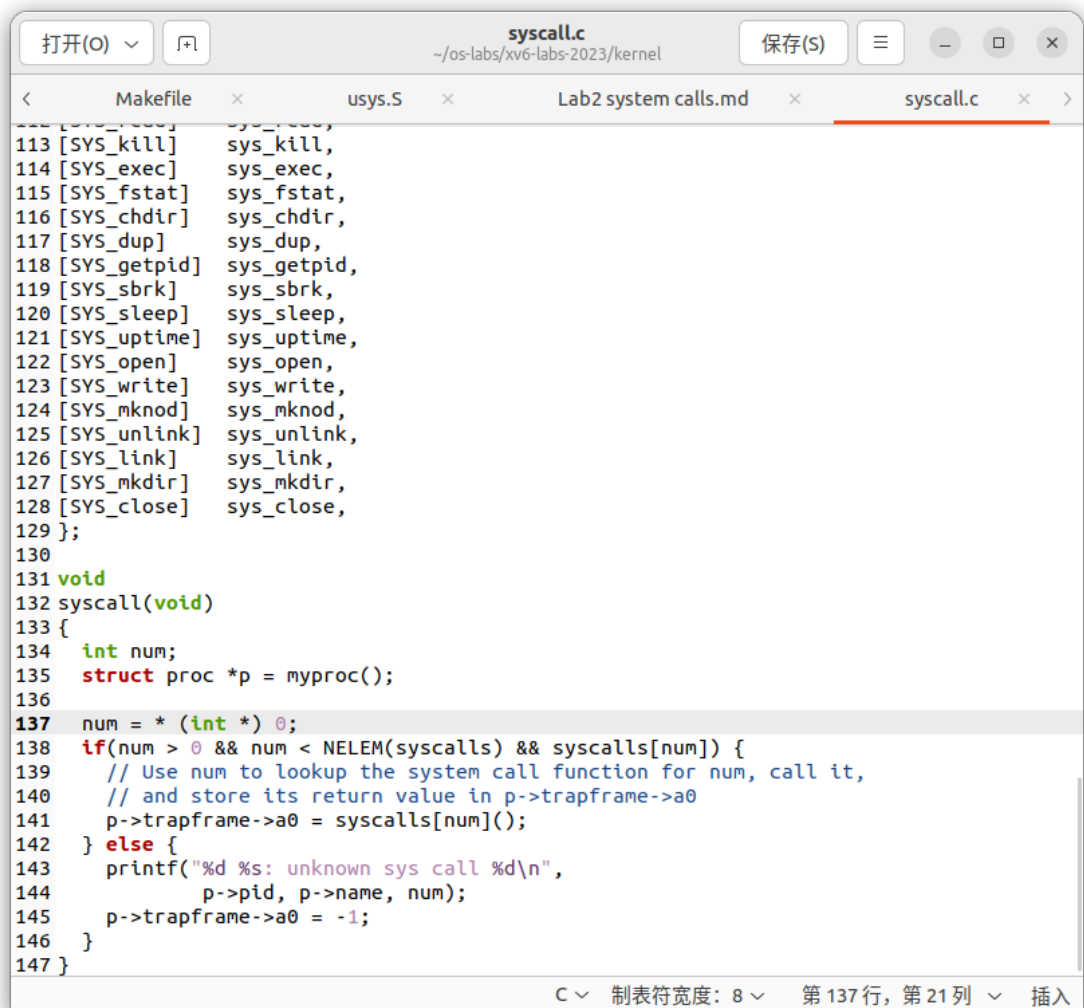
- 位 5 (**SIE**) = **1**：表示启用了监督模式下的中断。
- 位 1 (**SPP**) = **0**：表示 CPU 在进入当前模式之前处于用户模式。
- 其他位为 **0**。

结论

根据 **SPP** 位的值 **0**，CPU 之前处于 用户模式 (User Mode)。当前 CPU 正处于监督模式 (Supervisor Mode)，因为 **SIE** 位启用了中断。

这意味着程序执行发生了模式切换，例如从用户模式切换到内核模式（通过系统调用或异常），CPU 进入了监督模式以处理系统调用或异常。在内核模式下，**sstatus** 显示之前的模式为用户模式，这就是 **SPP** 位为 **0** 的原因。

- **Q3: What was the previous mode that the CPU was in?**
- 用户模式 (User Mode)。
- 将位于 **kernel/syscall.c** 中的 **syscall** 函数修改，替换 **syscall()** 函数中的 **num = p->trapframe->a7;** 为 **num = * (int *) 0;**，然后运行 **make qemu**。这样会看到一个 panic 信息。



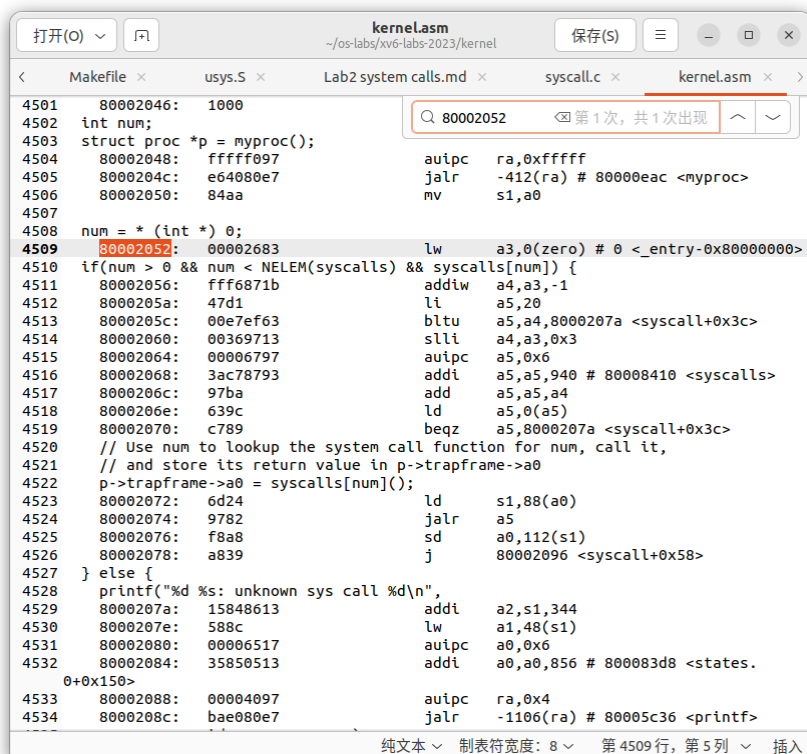
```
113 [SYS_kill] sys_kill,
114 [SYS_exec] sys_exec,
115 [SYS_fstat] sys_fstat,
116 [SYS_chdir] sys_chdir,
117 [SYS_dup] sys_dup,
118 [SYS_getpid] sys_getpid,
119 [SYS_sbrk] sys_sbrk,
120 [SYS_sleep] sys_sleep,
121 [SYS_uptime] sys_uptime,
122 [SYS_open] sys_open,
123 [SYS_write] sys_write,
124 [SYS_mknod] sys_mknod,
125 [SYS_unlink] sys_unlink,
126 [SYS_link] sys_link,
127 [SYS_mkdir] sys_mkdir,
128 [SYS_close] sys_close,
129 };
130
131 void
132 syscall(void)
133 {
134     int num;
135     struct proc *p = myproc();
136
137     num = * (int *) 0;
138     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
139         // Use num to lookup the system call function for num, call it,
140         // and store its return value in p->trapframe->a0
141         p->trapframe->a0 = syscalls[num]();
142     } else {
143         printf("%d %s: unknown sys call %d\n",
144             p->pid, p->name, num);
145         p->trapframe->a0 = -1;
146     }
147 }
```

```
chengyu@chengyu-virtual-machine: ~/os-labs/xv6-labs-2023
lib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/syscall.o
kernel/syscall.c
riscv64-linux-gnu-ld -z max-page-size=4096 -T kernel/kernel.ld -o kernel/kernel
kernel/entry.o kernel/kalloc.o kernel/string.o kernel/main.o kernel/vm.o kernel/
proc.o kernel/swtch.o kernel/trampoline.o kernel/trap.o kernel/syscall.o kernel/
sysproc.o kernel/bio.o kernel/fs.o kernel/log.o kernel/sleeplock.o kernel/file.o
kernel/pipe.o kernel/exec.o kernel/sysfile.o kernel/kernelvec.o kernel/plic.o k
kernel/virtio_disk.o kernel/start.o kernel/console.o kernel/printf.o kernel/uart.
o kernel/spinlock.o
riscv64-linux-gnu-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-linux-gnu-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /;
/^$/d' > kernel/kernel.sym
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp
3 -nographic -global virtio-mmio.force-legacy=false -drive file=fs.img,if=none,f
ormat=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
scause 0x000000000000000d
sepc=0x0000000080002052 stval=0x0000000000000000
panic: kerneltrap
```

- 这里的 `sepc` 指代内核发生 panic 的代码地址。可以在 `kernel/kernel.asm` 中查看编译后的完整内核汇编代码，在其中搜索这个地址既可以找到使内核 panic 的代码。`sepc` 的值不是固定不变的。



```
kernel.asm
~/os-labs/xv6-labs-2023/kernel
保存(S)
Makefile x usys.S x Lab2 system calls.md x syscall.c x kernel.asm x
4501 80002046: 1000
4502 int num;
4503 struct proc *p = myproc();
4504 80002048: fffff097 auipc ra,0xfffff
4505 8000204c: e64080e7 jalr -412(ra) # 80000eac <myproc>
4506 80002050: 84aa mv s1,a0
4507
4508 num = * (int *) 0;
4509 80002052: 00002683 lw a3,0(zero) # 0 <_entry-0x80000000>
4510 if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
4511 80002056: fff6871b addiw a4,a3,-1
4512 8000205a: 47d1 li a5,20
4513 8000205c: 00e7ef63 bltu a5,a4,8000207a <syscall+0x3c>
4514 80002060: 00369713 slli a4,a3,0x3
4515 80002064: 00006797 auipc a5,0x6
4516 80002068: 3ac78793 addi a5,a5,940 # 80008410 <syscalls>
4517 8000206c: 97ba add a5,a5,a4
4518 8000206e: 639c ld a5,0(a5)
4519 80002070: c789 beqz a5,8000207a <syscall+0x3c>
4520 // Use num to lookup the system call function for num, call it,
4521 // and store its return value in p->trapframe->a0
4522 p->trapframe->a0 = syscalls[num]();
4523 80002072: 6d24 ld s1,88(a0)
4524 80002074: 9782 jalr a5
4525 80002076: f8a8 sd a0,112(s1)
4526 80002078: a839 j 80002096 <syscall+0x58>
4527 } else {
4528 printf("%d %s: unknown sys call %d\n",
4529 8000207a: 15848613 addi a2,s1,344
4530 8000207e: 588c lw a1,48(s1)
4531 80002080: 00006517 auipc a0,0x6
4532 80002084: 35850513 addi a0,a0,856 # 800083d8 <states.
0+0x150>
4533 80002088: 00004097 auipc ra,0x4
4534 8000208c: bae080e7 jalr -1106(ra) # 80005c36 <printf>
```

- 可以看到，果然是 `num = * (int *) 0;` 使内核 panic。对应的汇编则是 `lw a3,0(zero)`。

- 所以这条汇编代码代表：将内存中地址从 0 开始的一个字 word（2 bytes）大小的数据加载到寄存器 `a3` 中。
- Q4: Write down the assembly instruction the kernel is panicing at. Which register corresponds to the variable `num`?
- A: 内核 panic 在 `lw a3,0(zero)`。 `num` 代表 `a3` 寄存器。
- 再次运行虚拟器和 GDB 调试。将断点设置在发生 panic 处。
- 再次输入 `n` 之后会发生 panic，此时输入 `Ctrl + C` 结束。查看 `scause` 寄存器，它代指内核 panic 的原因，查看文档 [RISC-V privileged instructions 4.1.8 章节](#)。

```

chengyu@chengyu-virtual-machine: ~/os-labs/xv6-labs-2023
0x80005c1e <panic+50> addi    a0,a0,1070
0x80005c22 <panic+54> auipc   ra,0x0
0x80005c26 <panic+58> jalr    20(ra)
0x80005c2a <panic+62> li      a5,1
0x80005c2c <panic+64> auipc   a4,0x3
0x80005c30 <panic+68> sw      a5,-800(a4)
> 0x80005c34 <panic+72> j       0x80005c34 <panic+72>
0x80005c36 <printf>    addi    sp,sp,-192
0x80005c38 <printf+2>    sd      ra,120(sp)
0x80005c3a <printf+4>    sd      s0,112(sp)
0x80005c3c <printf+6>    sd      s1,104(sp)
0x80005c3e <printf+8>    sd      s2,96(sp)
0x80005c40 <printf+10> sd      s3,88(sp)

remote Thread 1.2 In: panic L126 PC: 0x80005c34
(gdb) n

Thread 2 received signal SIGINT, Interrupt.
panic (s=s@entry=0x800083c0 "kerneltrap") at kernel/printf.c:126
(gdb) p $scause
$1 = 13
(gdb)

```

- 所以这里的 `13` 代表 `Load page fault`。就是从内存地址 0 中加载数据到寄存器 `a3` 时出错。

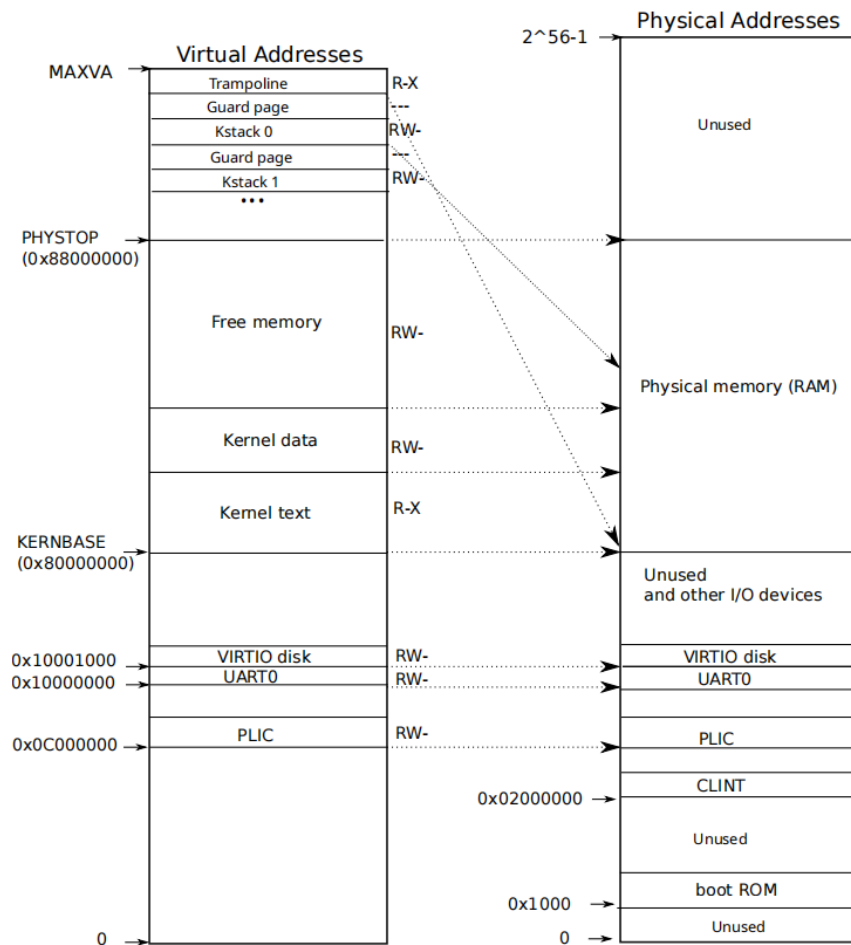
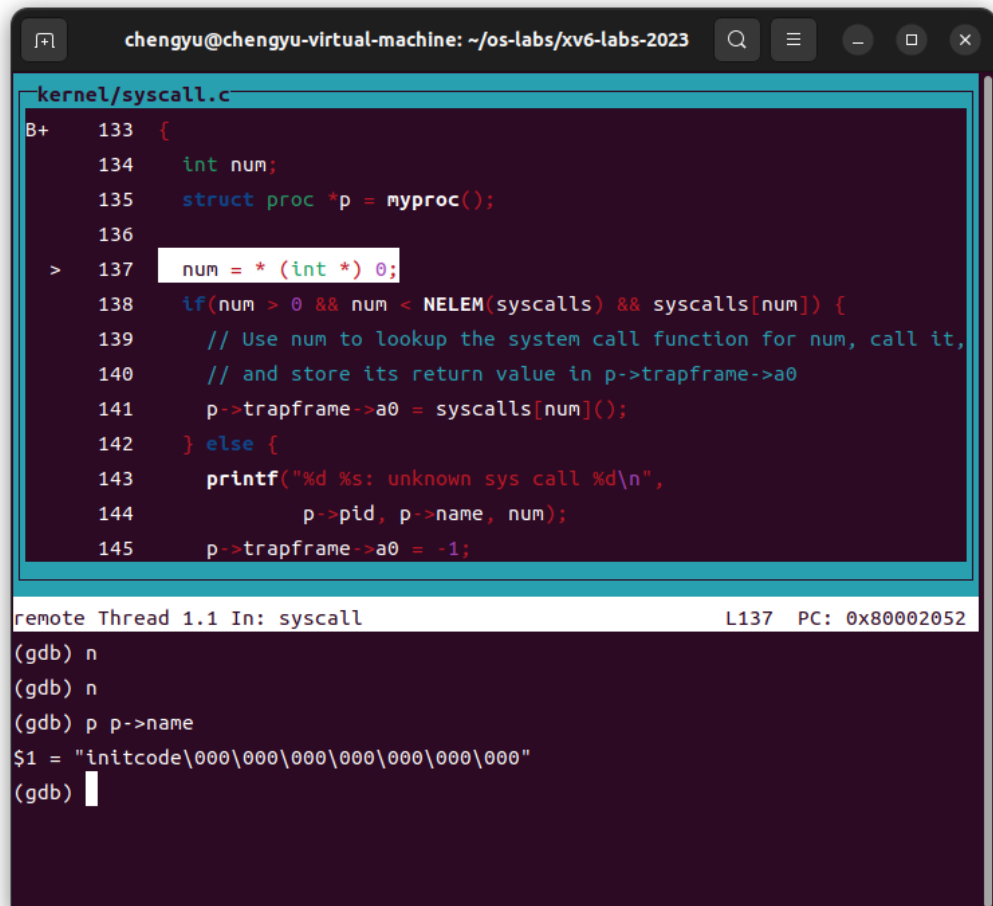


Figure 3.3: On the left, xv6's kernel address space. RWX refer to PTE read, write, and execute permissions. On the right, the RISC-V physical address space that xv6 expects to see.

- 可以看到，在左侧 Virtual Address 中的地址 0 处对应右侧 Physical Address 的 Unused，表示这个地址没有被使用。而 Kernel 是从虚拟地址的 `0x80000000` 处开始的。
- **Q5: Why does the kernel crash? Hint: look at figure 3-3 in the text; is address 0 mapped in the kernel address space? Is that confirmed by the value in `scause` above? (See description of `scause` in RISC-V privileged instructions)**
- **A:** 内核因为加载了一个未使用的地址 0 处的内存数据而崩溃（Load page fault）。地址 0 并不映射到内核空间中（从 `0x80000000` 开始）。`scause` 中的异常代码证实了上述观点。
- 上述 `scuase` 指明了内核 panic 的原因。但是有时候我们需要知道，是哪一个用户程序调用 `syscall` 时发生了 panic。这可以通过打印 `proc` 结构体中的 `name` 来查看。
- 重新启动 qemu 和 gdb。
- 输入下列命令

```
1 (gdb) b syscall
2 (gdb) c
3 (gdb) layout src
4 (gdb) n
5 (gdb) n
6 (gdb) p p->name
```



```
chengyu@chengyu-virtual-machine: ~/os-labs/xv6-labs-2023
kernel/syscall.c
B+ 133 {
    134     int num;
    135     struct proc *p = myproc();
    136
    > 137     num = * (int *) 0;
    138     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    139         // Use num to lookup the system call function for num, call it,
    140         // and store its return value in p->trapframe->a0
    141         p->trapframe->a0 = syscalls[num]();
    142     } else {
    143         printf("%d %s: unknown sys call %d\n",
    144             p->pid, p->name, num);
    145         p->trapframe->a0 = -1;
    146     }
    147 }

remote Thread 1.1 In: syscall L137 PC: 0x80002052
(gdb) n
(gdb) n
(gdb) p p->name
$1 = "initcode\000\000\000\000\000\000\000"
(gdb)
```

- 可以看到，这个用户程序是 `initcode`，也是 xv6 第一个 process。

打印 `proc` 结构体可以查看这个进程的其他信息。

```
chengyu@chengyu-virtual-machine: ~/os-labs/xv6-labs-2023
kernel/syscall.c
B+ 133 {
    134     int num;
    135     struct proc *p = myproc();
    136
    > 137     num = * (int *) 0;
    138     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    139         // Use num to lookup the system call function for num, call it,
    140         // and store its return value in p->trapframe->a0
    141         p->trapframe->a0 = syscalls[num]();
    142     } else {
    143         printf("%d %s: unknown sys call %d\n",
    144             p->pid, p->name, num);
    145         p->trapframe->a0 = -1;
    }

remote Thread 1.1 In: syscall L137 PC: 0x80002052
$2 = {lock = {locked = 0, name = 0x800081b8 "proc", cpu = 0x0},
    state = RUNNING, chan = 0x0, killed = 0, xstate = 0, pid = 1, parent = 0x0,
    kstack = 274877894656, sz = 4096, pagetable = 0x87f73000,
    trapframe = 0x87f74000, context = {ra = 2147488962, sp = 274877898336,
    s0 = 274877898384, s1 = 2147519856, s2 = 2147518784, s3 = 0,
    s4 = 361700864190383365, s5 = 361700864190383365, s6 = 361700864190383365,
    s7 = 361700864190383365, s8 = 361700864190383365, s9 = 361700864190383365,
    --Type <RET> for more, q to quit, c to continue without paging--
```

- 可以看到，这个 `initcode` 的 pid 为 1。
- Q6: What is the name of the binary that was running when the kernel panicked?
What is its process id (pid)?
- A: 这个二进制的名字为 `initcode`，其 process id 为 1。

实验中遇到的问题

问题描述

- 暂无。

解决方案

- 暂无。

2. System call tracing (moderate)

实验目的

- 此任务会增加一个系统调用追踪功能，它将会在后续实验的调试时有所帮助。课程提供了一个 `trace` 程序，它将会运行并开始另一个程序的系统调用追踪功能（tracing enable），此程序位于 `user/trace.c`。其参数为一个掩码 `mask`，用来指示其要追踪的系统调用。例如 `trace(1 << SYS_fork)`，`SYS_fork` 为系统调用号在文件 `kernel/syscall.h` 中。如果系统调用号被设置在掩码中，你必须修改 `xv6` 内核，当每一个追踪的系统调用将要返回的时候打印一行信息。这一行信息包含进程 `id`，系统调用的名字和要返回的值。你不需要打印系统调用的参数。`trace` 系统调用应该启用它调用的程序和它调用程序的每一个子程序的追踪功能，但是不能影响其他进程。

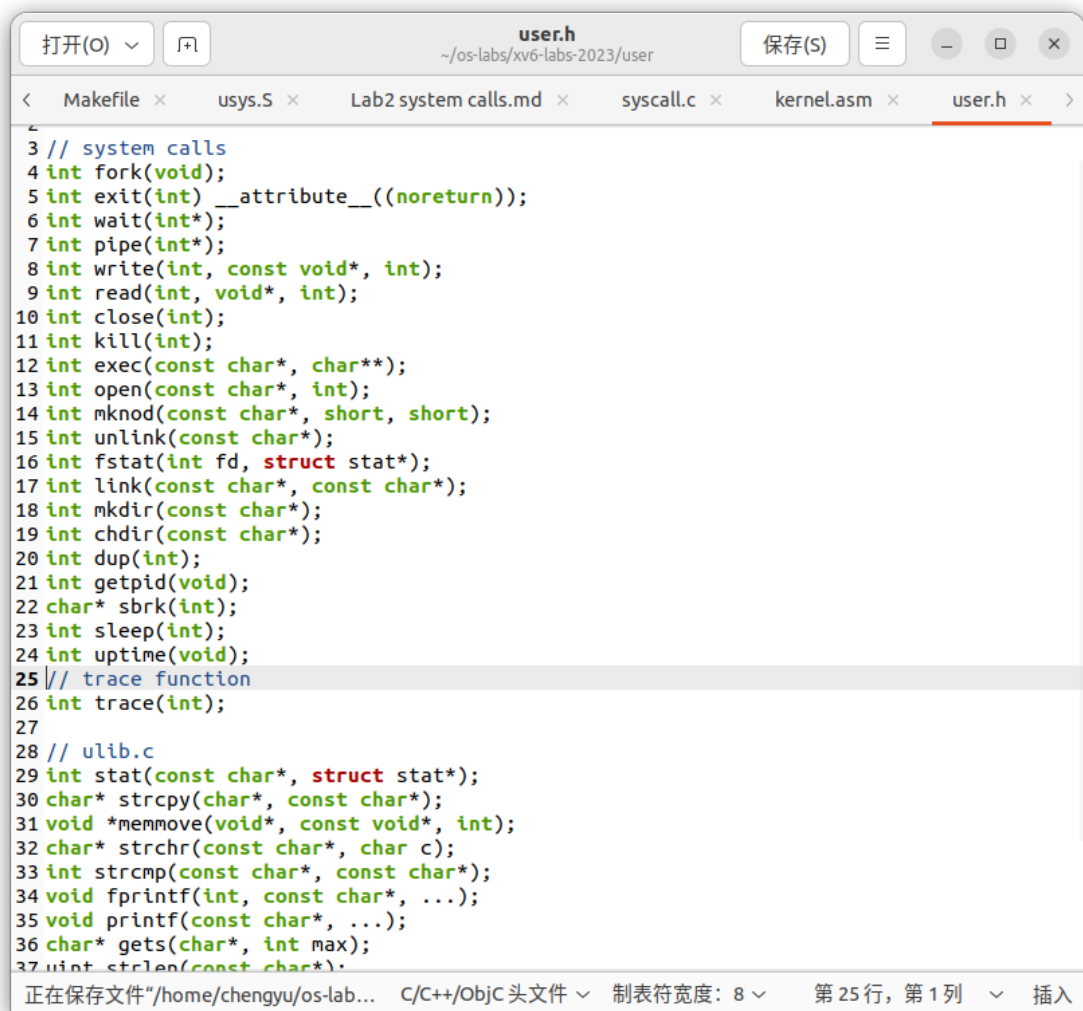
实验步骤

添加到 `Makefile`

- 将 `$U/_trace` 添加到 `Makefile` 的 `UPROGS` 中。

添加 `trace` 系统调用的定义

- 在 `user/user.h` 中添加这个系统调用的函数原型；



The screenshot shows a code editor window titled "user.h" with the path "~/os-labs/xv6-labs-2023/user". The editor displays a list of system call prototypes in C. The line numbers range from 3 to 37. The code is as follows:

```
3 // system calls
4 int fork(void);
5 int exit(int) __attribute__((noreturn));
6 int wait(int*);
7 int pipe(int*);
8 int write(int, const void*, int);
9 int read(int, void*, int);
10 int close(int);
11 int kill(int);
12 int exec(const char*, char**);
13 int open(const char*, int);
14 int mknod(const char*, short, short);
15 int unlink(const char*);
16 int fstat(int fd, struct stat*);
17 int link(const char*, const char*);
18 int mkdir(const char*);
19 int chdir(const char*);
20 int dup(int);
21 int getpid(void);
22 char* sbrk(int);
23 int sleep(int);
24 int uptime(void);
25 // trace function
26 int trace(int);
27
28 // ulib.c
29 int stat(const char*, struct stat*);
30 char* strcpy(char*, const char*);
31 void *memmove(void*, const void*, int);
32 char* strchr(const char*, char c);
33 int strcmp(const char*, const char*);
34 void fprintf(int, const char*, ...);
35 void printf(const char*, ...);
36 char* gets(char*, int max);
37 uint strlen(const char*);
```

The status bar at the bottom indicates "正在保存文件"/home/chengyu/os-lab... C/C++/ObjC 头文件 制表符宽度: 8 第 25 行, 第 1 列 插入".

- 在 `user/usys.pl` 中添加一个 `entry`，它将会生成 `user/usys.S`，里面包含真实的汇编代码，它使用 Risc V 的 `ecall` 指令陷入内核，执行系统调用；

```
5 print "# generated by usys.pl - do not edit\n";
6
7 print "#include \"kernel/syscall.h\"\n";
8
9 sub entry {
10     my $name = shift;
11     print ".global $name\n";
12     print "${name}:\n";
13     print "li a7, SYS_${name}\n";
14     print "ecall\n";
15     print "ret\n";
16 }
17
18 entry("fork");
19 entry("exit");
20 entry("wait");
21 entry("pipe");
22 entry("read");
23 entry("write");
24 entry("close");
25 entry("kill");
26 entry("exec");
27 entry("open");
28 entry("mknod");
29 entry("unlink");
30 entry("fstat");
31 entry("link");
32 entry("mkdir");
33 entry("chdir");
34 entry("dup");
35 entry("getpid");
36 entry("sbrk");
37 entry("sleep");
38 entry("uptime");
39 entry("trace");
```

- 在 `kernel/syscall.h` 中添加一个系统调用号;

```
1 // System call numbers
2 #define SYS_fork 1
3 #define SYS_exit 2
4 #define SYS_wait 3
5 #define SYS_pipe 4
6 #define SYS_read 5
7 #define SYS_kill 6
8 #define SYS_exec 7
9 #define SYS_fstat 8
10 #define SYS_chdir 9
11 #define SYS_dup 10
12 #define SYS_getpid 11
13 #define SYS_sbrk 12
14 #define SYS_sleep 13
15 #define SYS_uptime 14
16 #define SYS_open 15
17 #define SYS_write 16
18 #define SYS_mknod 17
19 #define SYS_unlink 18
20 #define SYS_link 19
21 #define SYS_mkdir 20
22 #define SYS_close 21
23 #define SYS_trace 22
```

添加 `sys_trace()`

- 在 `kernel/sysproc.c` 中添加一个 `sys_trace()` 函数作为系统调用。

```
1 //add trace
2
3 uint64
4 sys_trace()
5 {
6     int mask;
7
8     argint(0, &mask);
9
10    struct proc *pro = myproc();
11    printf("trace pid: %d\n", pro->pid);
12    pro->trace_mask = mask;
13
14    return 0;
15 }
```


修改 struct proc

- 添加 trace_mask 变量

```
1 // these are private to the process, so p->lock need not be
  held.
2  uint64 kstack;           // virtual address of kernel
  stack
3  uint64 sz;               // Size of process memory (bytes)
4  pagetable_t pagetable;   // User page table
5  struct trapframe *trapframe; // data page for trampoline.S
6  struct context context;   // swtch() here to run process
7  struct file *ofile[NOFILE]; // Open files
8  struct inode *cwd;        // Current directory
9  char name[16];           // Process name (debugging)
10 int trace_mask;           // trace syscall mas
```

修改 syscall.c

```
1 extern uint64 sys_trace(void);
2
3 // An array mapping syscall numbers from syscall.h
4 // to the function that handles the system call.
5 static char *syscall_name[] = {
6     "", "fork", "exit", "wait", "pipe", "read", "kill",
7     "exec", "fstat", "chdir", "dup",
8     "getpid", "sbrk", "sleep", "uptime", "open", "write",
9     "mknod", "unlink", "link", "mkdir",
10    "close", "trace"
11 };
12
13 void
14 syscall(void)
15 {
16     int num;
17     struct proc *p = myproc();
18
19     num = p->trapframe->a7;
20     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
21         p->trapframe->a0 = syscalls[num]();
22     }
```

```

21     if (p->trace_mask & (1 << num)) {
22         printf("%d: syscall %s -> %d\n", p->pid,
syscall_name[num], p->trapframe->a0);
23     }
24 } else {
25     printf("%d %s: unknown sys call %d\n",
26         p->pid, p->name, num);
27     p->trapframe->a0 = -1;
28 }
29 }

```

清除掩码

- 进程清除时也应清除相应掩码 `proc.c/freeproc`:

```

1  // free a proc structure and the data hanging from it,
2  // including user pages.
3  // p->lock must be held.
4  static void
5  freeproc(struct proc *p)
6  {
7      if(p->trapframe)
8          kfree((void*)p->trapframe);
9      p->trapframe = 0;
10     if(p->pagetable)
11         proc_freepagetable(p->pagetable, p->sz);
12     p->pagetable = 0;
13     p->sz = 0;
14     p->pid = 0;
15     p->parent = 0;
16     p->name[0] = 0;
17     p->chan = 0;
18     p->killed = 0;
19     p->xstate = 0;
20     p->state = UNUSED;
21
22     // trace
23     p->trace_mask = 0;
24 }

```

修改fork

- fork时子进程也复制到该掩码 `proc.c/fork`:

```
1 // Create a new process, copying the parent.
2 // Sets up child kernel stack to return as if from fork() system
  call.
3 int fork(void) {
4     int i, pid;
5     struct proc *np;
6     struct proc *p = myproc();
7
8     // Allocate process.
9     if ((np = allocproc()) == 0) {
10         return -1;
11     }
12
13     // Copy user memory from parent to child.
14     if (uvmcopy(p->pagetable, np->pagetable, p->sz) < 0) {
15         freeproc(np);
16         release(&np->lock);
17         return -1;
18     }
19     np->sz = p->sz;
20
21     // Copy trace mask value
22     np->trace_mask = p->trace_mask;
23
24     // Copy saved user registers.
25     *(np->trapframe) = *(p->trapframe);
26
27     // Cause fork to return 0 in the child.
28     np->trapframe->a0 = 0;
29
30     // Increment reference counts on open file descriptors.
31     for (i = 0; i < NOFILE; i++) {
32         if (p->ofile[i]) {
33             if ((np->ofile[i] = filedup(p->ofile[i])) == 0) {
34                 // If filedup fails, cleanup and exit
35                 while (--i >= 0) {
36                     if (np->ofile[i]) {
37                         fcloselose(np->ofile[i]);
```

```

38         }
39     }
40     freeproc(np);
41     release(&np->lock);
42     return -1;
43 }
44 }
45 }
46
47 // Duplicate the current working directory.
48 if ((np->cwd = idup(p->cwd)) == 0) {
49     freeproc(np);
50     release(&np->lock);
51     return -1;
52 }
53
54 // Copy process name
55 safestrcpy(np->name, p->name, sizeof(p->name));
56
57 // Assign pid and set process to runnable
58 pid = np->pid;
59
60 release(&np->lock);
61
62 // Set parent process and update wait_lock
63 acquire(&wait_lock);
64 np->parent = p;
65 release(&wait_lock);
66
67 // Finally, set the child process to runnable
68 acquire(&np->lock);
69 np->state = RUNNABLE;
70 release(&np->lock);
71
72 return pid;
73 }

```

测试结果

```
chengyu@chengyu-virtual-machine: ~/os-labs/xv6-labs-2023
ALL -DLAB_SYSCALL -MD -mcmodel=medany -ffreestanding -fno-common -nostdlib -mno-relax -I.
-fno-stack-protector -fno-pie -no-pie -c -o kernel/spinlock.o kernel/spinlock.c
riscv64-linux-gnu-ld -z max-page-size=4096 -T kernel/kernel.ld -o kernel/kernel kernel/en
try.o kernel/kalloc.o kernel/string.o kernel/main.o kernel/vm.o kernel/proc.o kernel/swtc
h.o kernel/trampoline.o kernel/trap.o kernel/syscall.o kernel/sysproc.o kernel/bio.o kern
el/fs.o kernel/log.o kernel/sleeplock.o kernel/file.o kernel/pipe.o kernel/exec.o kernel/
sysfile.o kernel/kernelvec.o kernel/plic.o kernel/virtio_disk.o kernel/start.o kernel/con
sole.o kernel/printf.o kernel/uart.o kernel/spinlock.o
riscv64-linux-gnu-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-linux-gnu-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' >
kernel/kernel.sym
== Test trace 32 grep == trace 32 grep: OK (3.8s)
== Test trace all grep == trace all grep: OK (2.6s)
== Test trace nothing == trace nothing: OK (2.6s)
== Test trace children == Timeout! trace children: FAIL (30.5s)
...
8: syscall fork -> 22
7: syscall fork -> 23
6: syscall fork -> 24
10: syscall fork -> 25
qemu-system-riscv64: terminating on signal 15 from pid 52795 (make)
MISSING '^\\d+: syscall fork -> -1'
QEMU output saved to xv6.out.trace_children
chengyu@chengyu-virtual-machine: ~/os-labs/xv6-labs-2023$
```

```
chengyu@chengyu-virtual-machine: ~/os-labs/xv6-labs-2023
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -global virtio-mmio.force-legacy=false -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive
=x0,bus=virtio-mmio-bus.0
xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ trace 32 grep hello README
trace pid: 3
3: syscall read -> 1023
3: syscall read -> 961
3: syscall read -> 321
3: syscall read -> 0
$ trace 2147483647 grep hello README
trace pid: 4
4: syscall trace -> 0
4: syscall exec -> 3
4: syscall open -> 3
4: syscall read -> 1023
4: syscall read -> 961
4: syscall read -> 321
4: syscall read -> 0
4: syscall close -> 0
$ grep hello README
$ trace 2 usertests forkforkfork
trace pid: 6
usertests starting
6: syscall fork -> 7
test forkforkfork: 6: syscall fork -> 8
8: syscall fork -> 9
9: syscall fork -> 10
9: syscall fork -> 11
10: syscall fork -> 12
9: syscall fork -> 13
11: syscall fork -> 14
10: syscall fork -> 15
9: syscall fork -> 16
11: syscall fork -> 17
10: syscall fork -> 18
9: syscall fork -> 19
11: syscall fork -> 20
```

```
chengyu@chengyu-virtual-machine: ~/os-labs/xv6-labs-2023
11: syscall fork -> 53
20: syscall fork -> 54
23: syscall fork -> 55
12: syscall fork -> 56
14: syscall fork -> 57
13: syscall fork -> 58
9: syscall fork -> 59
11: syscall fork -> 60
10: syscall fork -> 61
12: syscall fork -> 62
9: syscall fork -> 63
10: syscall fork -> 64
59: syscall fork -> 65
27: syscall fork -> 66
9: syscall fork -> 67
10: syscall fork -> 68
11: syscall fork -> -1
9: syscall fork -> -1
10: syscall fork -> -1
40: syscall fork -> -1
OK
6: syscall fork -> 69
ALL TESTS PASSED
$ $
```

实验中遇到的问题

问题描述

1. 系统调用返回值未打印：在最初的实现中，系统调用返回值没有被正确打印，导致调试信息不完整，无法准确判断系统调用是否成功。
2. 掩码传播问题：在实现 `fork` 时，子进程没有正确继承父进程的追踪掩码，导致子进程的系统调用未被追踪。
3. 文件描述符管理问题：在使用 `printf` 打印调试信息时，出现了文件描述符资源未正确管理的问题，导致部分系统调用的输出被截断或缺失。

解决方案

1. 修正返回值打印：通过在 `syscall()` 函数中确保在调用完系统调用函数后立即打印返回值信息，解决了返回值未打印的问题。
2. 正确传播掩码：在 `fork()` 实现中，添加代码确保子进程正确继承父进程的 `trace_mask`，从而使得子进程的系统调用也能被追踪到。
3. 优化文件描述符管理：仔细检查了 `printf` 使用的文件描述符，并确保每次调试信息打印后，文件描述符都能正确关闭，从而避免了输出问题。

实验心得

通过本次实验，我深入了解了 xv6 系统中系统调用的工作原理，并掌握了如何添加和管理系统调用的追踪功能。这次实验让我认识到，在操作系统中实现调试功能时，细节管理尤为重要，例如确保系统调用的返回值正确打印、子进程继承父进程的掩码等。

在实际操作中，我遇到了返回值未正确打印、掩码未正确传播等问题，但通过分析和修正代码，最终实现了预期的系统调用追踪功能。这次实验不仅提升了我对 xv6 内核结构的理解，也让我在调试和解决问题的过程中积累了宝贵的经验。通过这种方式，我加深了对系统编程的理解，为后续实验奠定了坚实的基础。

3. Sysinfo(moderate)

实验目的

- 在本次实验中，你将添加一个 `sysinfo` 系统调用，用于收集当前运行系统的信息。该系统调用接受一个参数：指向 `struct sysinfo`（参见 `kernel/sysinfo.h`）的指针。内核应填写此结构体的字段：`freemem` 字段应设置为系统中空闲内存的字节数，`nproc` 字段应设置为状态不是 `UNUSED` 的进程数。我们提供了一个测试程序 `sysinfotest`；当该程序打印出 "sysinfotest: OK" 时，说明你已通过此实验。

实验步骤

添加到 Makefile

- 将 `$U/_sysinfotest` 添加到 `UPROGS` 中的 `Makefile`：
 - 打开 `Makefile`，找到 `UPROGS` 行，将 `user/sysinfotest.c` 对应的目标文件添加进去。

添加 `sysinfo` 系统调用的定义

- 在 `user/user.h` 中添加这个系统调用的函数原型；
- 在 `user/usys.pl` 中添加一个 `entry`，它将会生成 `user/usys.S`，里面包含真实的汇编代码，它使用 Risc V 的 `ecall` 指令陷入内核，执行系统调用；

```
1  entry("fork");
2  entry("exit");
3  entry("wait");
4  entry("pipe");
5  entry("read");
6  entry("write");
7  entry("close");
8  entry("kill");
9  entry("exec");
10 entry("open");
11 entry("mknod");
12 entry("unlink");
13 entry("fstat");
14 entry("link");
15 entry("mkdir");
16 entry("chdir");
17 entry("dup");
18 entry("getpid");
19 entry("sbrk");
20 entry("sleep");
21 entry("uptime");
22 entry("trace");
23 entry("sysinfo");
```

- 在 `kernel/syscall.h` 中添加一个系统调用号；

```
1  // system call numbers
2  #define SYS_fork    1
3  #define SYS_exit    2
4  #define SYS_wait    3
5  #define SYS_pipe    4
6  #define SYS_read    5
7  #define SYS_kill    6
8  #define SYS_exec    7
9  #define SYS_fstat   8
10 #define SYS_chdir   9
11 #define SYS_dup     10
12 #define SYS_getpid  11
13 #define SYS_sbrk    12
14 #define SYS_sleep   13
15 #define SYS_uptime  14
```



```
16 #define SYS_open 15
17 #define SYS_write 16
18 #define SYS_mknod 17
19 #define SYS_unlink 18
20 #define SYS_link 19
21 #define SYS_mkdir 20
22 #define SYS_close 21
23 #define SYS_trace 22
24 #define SYS_sysinfo 23
```

- 修改 `kernel/syscall.c`

```
1 extern uint64 sys_sysinfo(void);
2
3 [SYS_sysinfo] sys_sysinfo,
```

修改 `kernel/kalloc.c`

- 在 `kernel/kalloc.c` 中添加一个函数用于计算未使用的空闲内存。

```
1 // 统计未使用内存
2 // 一页等于 4096 bytes
3 uint64
4 free_mem_num(void)
5 {
6
7     struct run *r;
8     uint64 free_num = 0;
9     acquire(&kmem.lock);
10    r = kmem.freelist;
11    while (r) {
12        free_num++;
13        r = r->next;
14    }
15    release(&kmem.lock);
16    return free_num * 4096;
17 }
```

修改 kernel/proc.c

- 在 kernel/proc.c 中添加一个函数用于收集进程数量。

```
1 // used by sysinfo
2 int
3 proc_not_unused_num(void)
4 {
5     int nproc = 0;
6     for (struct proc *p = proc; p < &proc[NPROC]; p++) {
7         if (p->state != UNUSED)
8             nproc++;
9     }
10    return nproc;
11 }
```

将实现的两个函数的定义添加到 kernel/defs.h 中

```
1 // used by sysinfo
2 int proc_not_unused_num(void);
```

```
1 // used by sysinfo
2 uint64 free_mem_num(void);
```

在 kernel/sysproc.c 中实现 sysinfo 系统调用

```
1 uint64
2 sys_sysinfo(void)
3 {
4     // user pointer to struct sysinfo
5     uint64 si_addr;
6
7     argaddr(0, &si_addr);
8     int nproc;
9     int freemem;
10
11    nproc = proc_not_unused_num();
12    freemem = free_mem_num();
13
14    struct sysinfo sysinfo;
```

```

15     sysinfo.freemem = freemem;
16     sysinfo.nproc = nproc;
17
18     struct proc *p = myproc();
19     if (copyout(p->pagetable, si_addr, (char *)&sysinfo,
20         sizeof(sysinfo)) < 0)
21         return -1;
22     return 0;
23 }

```

- 注意在文件上方加上相关头文件：

```
1 #include "sysinfo.h"
```

涉及到的文件

```

chengyu@chengyu-virtual-machine: ~/os-labs/xv6-labs-2023
位于分支 syscall
您的分支与上游分支 'origin/syscall' 一致。

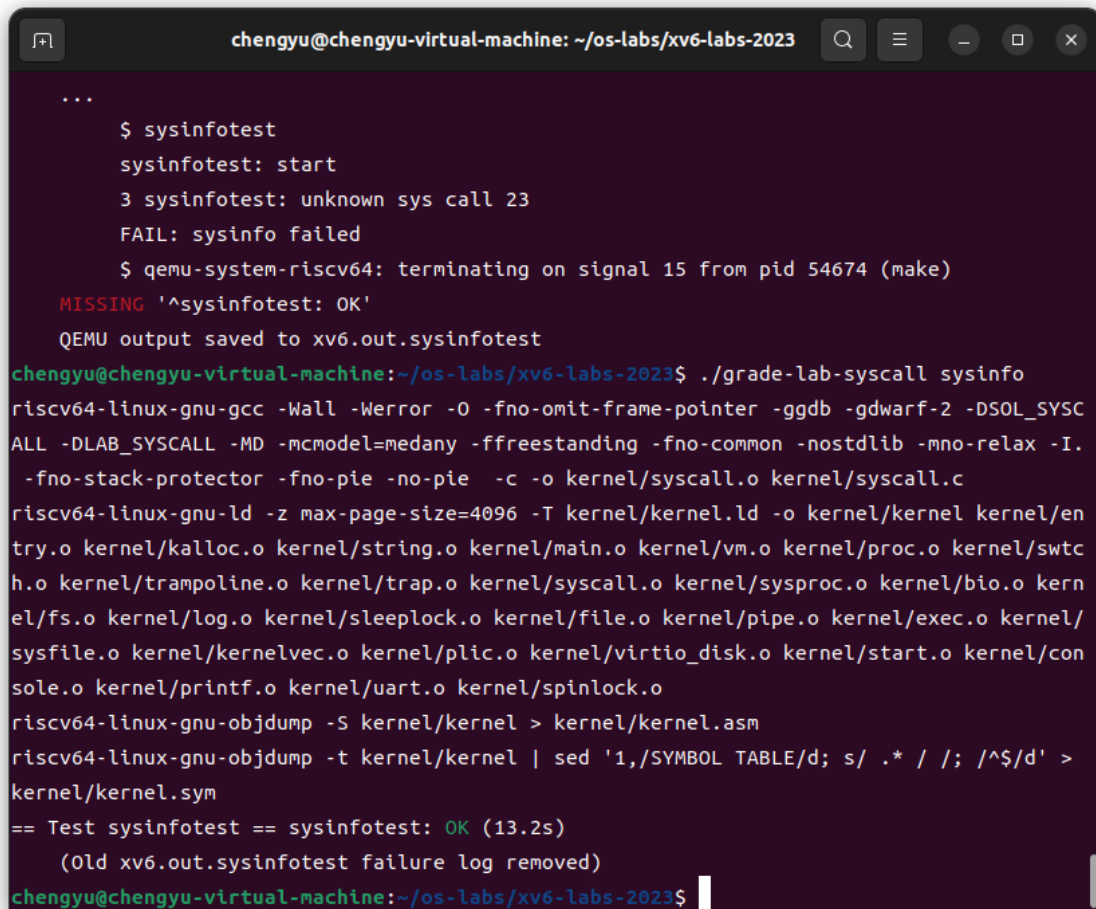
尚未暂存以备提交的变更:
(使用 "git add <文件>..." 更新要提交的内容)
(使用 "git restore <文件>..." 丢弃工作区的改动)
修改:      Makefile
修改:      kernel/defs.h
修改:      kernel/kalloc.c
修改:      kernel/proc.c
修改:      kernel/syscall.c
修改:      kernel/syscall.h
修改:      kernel/sysproc.c
修改:      user/user.h
修改:      user/usys.pl

未跟踪的文件:
(使用 "git add <文件>..." 以包含要提交的内容)
docs/img/add-sysinfo-in-user.h.png
docs/img/add-sysinfotest-to-Makefile.png
docs/img/test-sysinfo.png

修改尚未加入提交 (使用 "git add" 和/或 "git commit -a")
chengyu@chengyu-virtual-machine:~/os-labs/xv6-labs-2023$ git add -A

```

测试成功



```
chengyu@chengyu-virtual-machine: ~/os-labs/xv6-labs-2023
...
$ sysinfotest
sysinfotest: start
3 sysinfotest: unknown sys call 23
FAIL: sysinfo failed
$ qemu-system-riscv64: terminating on signal 15 from pid 54674 (make)
MISSING '^sysinfotest: OK'
QEMU output saved to xv6.out.sysinfotest
chengyu@chengyu-virtual-machine:~/os-labs/xv6-labs-2023$ ./grade-lab-syscall sysinfo
riscv64-linux-gnu-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_SYSCALL -DLAB_SYSCALL -MD -mmodel=medany -ffreestanding -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -no-pie -c -o kernel/syscall.o kernel/syscall.c
riscv64-linux-gnu-ld -z max-page-size=4096 -T kernel/kernel.ld -o kernel/kernel kernel/entry.o kernel/kalloc.o kernel/string.o kernel/main.o kernel/vm.o kernel/proc.o kernel/swtch.o kernel/trampoline.o kernel/trap.o kernel/syscall.o kernel/sysproc.o kernel/bio.o kernel/fs.o kernel/log.o kernel/sleeplock.o kernel/file.o kernel/pipe.o kernel/exec.o kernel/sysfile.o kernel/kernelvec.o kernel/plic.o kernel/virtio_disk.o kernel/start.o kernel/console.o kernel/printf.o kernel/uart.o kernel/spinlock.o
riscv64-linux-gnu-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-linux-gnu-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel/kernel.sym
== Test sysinfotest == sysinfotest: OK (13.2s)
(Old xv6.out.sysinfotest failure log removed)
chengyu@chengyu-virtual-machine:~/os-labs/xv6-labs-2023$
```

实验中遇到的问题

问题描述

1. 系统调用返回值未打印：在最初的实现中，系统调用返回值没有被正确打印，导致调试信息不完整，无法准确判断系统调用是否成功。
2. 掩码传播问题：在实现 `fork` 时，子进程没有正确继承父进程的追踪掩码，导致子进程的系统调用未被追踪。
3. 文件描述符管理问题：在使用 `printf` 打印调试信息时，出现了文件描述符资源未正确管理的问题，导致部分系统调用的输出被截断或缺失。

解决方案

1. 修正返回值打印：通过在 `syscall()` 函数中确保在调用完系统调用函数后立即打印返回值信息，解决了返回值未打印的问题。
2. 正确传播掩码：在 `fork()` 实现中，添加代码确保子进程正确继承父进程的 `trace_mask`，从而使得子进程的系统调用也能被追踪到。

3. 优化文件描述符管理：仔细检查了 `printf` 使用的文件描述符，并确保每次调试信息打印后，文件描述符都能正确关闭，从而避免了输出问题。

实验心得

通过本次实验，我深入了解了 `xv6` 系统中系统调用的工作原理，并掌握了如何添加和管理系统调用的追踪功能。这次实验让我认识到，在操作系统中实现调试功能时，细节管理尤为重要，例如确保系统调用的返回值正确打印、子进程继承父进程的掩码等。

在实际操作中，我遇到了返回值未正确打印、掩码未正确传播等问题，但通过分析和修正代码，最终实现了预期的系统调用追踪功能。这次实验不仅提升了我对 `xv6` 内核结构的理解，也让我在调试和解决问题的过程中积累了宝贵的经验。通过这种方式，我加深了对系统编程的理解，为后续实验奠定了坚实的基础。

实验得分

```
riscv64-linux-gnu-objdump -S kernel/kernel > kernel/kernel.asm
riscv64-linux-gnu-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel/kernel.sym
make[1]: Leaving directory '/home/yik/xv6-labs-2023'
== Test answers-syscall.txt ==
answers-syscall.txt: OK
== Test trace 32 grep ==
$ make qemu-gdb
trace 32 grep: OK (2.3s)
== Test trace all grep ==
$ make qemu-gdb
trace all grep: OK (0.9s)
== Test trace nothing ==
$ make qemu-gdb
trace nothing: OK (1.1s)
== Test trace children ==
$ make qemu-gdb
trace children: OK (13.6s)
== Test sysinfotest ==
$ make qemu-gdb
sysinfotest: OK (2.7s)
== Test time ==
time: OK
Score: 40/40
yik@LAPTOP-71JNVH53:~/xv6-labs-2023$ git checkout pgtbl
Branch 'pgtbl' set up to track remote branch 'pgtbl' from 'origin'.
```

- 初步诊断应该是 `fork()` 的实现有问题，之后努力改进。