

进程管理：

这个操作系统基于 xv6-riscv 和 xv6-k210, 进程管理部分实现了基本的进程创建、调度、终止和进程间通信等功能，下面将详细介绍这些功能的实现原理。

1. 进程的创建：

进程的创建是通过系统调用`fork`实现的。当一个进程调用`fork`时，操作系统会创建一个与当前进程完全相同的新进程，包括代码、数据和运行环境。这个新进程被称为子进程，而原始进程称为父进程。与 x86 架构不同的是，RISC-V 架构中`fork`系统调用需要保存更多的寄存器状态。在这个系统中，父进程需要保存一些特定的寄存器状态，然后将其复制到子进程的地址空间中，最后设置子进程的返回值寄存器为 0，从而实现了进程的创建。

2. 进程的调度：

这个系统同样采用时间片轮转的调度算法进行进程调度。时间片轮转算法将 CPU 时间划分成一个个时间片，每个进程在一个时间片内执行。当一个进程的时间片用完后，操作系统会将该进程挂起，将 CPU 分配给就绪队列中的下一个进程。为了使用定时器中断来触发时间片轮转，当定时器中断发生时，操作系统会保存当前进程的寄存器状态，切换到下一个进程执行，从而实现进程的调度。在切换进程时，需要保存当前进程的上下文，包括通用寄存器、程序计数器和其他特定寄存器，然后恢复下一个进程的上下文，从而实现进程的切换。

3. 进程的终止：

进程的终止是通过系统调用`exit`来实现的，其原理是当一个进程调用`exit`时，操作系统会释放该进程所占用的资源，包括内存空间和打开的文件等，并向其父进程发送一个信号，告知其已经终止。父进程可以通过调用`wait`系统调用等待子进程终止，并获取其退出状态。在等待子进程结束期间，父进程会被阻塞，直到子进程结束，从而避免了僵尸进程的产生。

4. 进程间通信：

这个系统实现了简单的进程间通信机制，包括管道(pipe)和共享内存(shared memory)。管道允许一个进程将输出传递给另一个进程，实现了进程间的单向通信。共享内存允许多个进程访问同一个内存区域，从而实现了进程间的共享数据。这些机制为不同进程之间的数据传递和协作提供了基础。

总体来说，系统实现了基本的进程创建、调度和终止功能，并提供了简单的进程间通信机制。在 RISC-V 架构下，进程的创建和切换需要保存和恢复更多的寄存器状态。进程管理部分是操作系统的核心部分，为多个进程的并发执行提供了基础。

系统调用：

这个系统实现了系统调用(syscall)机制，允许用户程序与操作系统内核进行交互。系统调用是用户程序与操作系统之间的接口，它允许用户程序请求操作系统提供特定的服务，如文件操作、进程管理、网络通信等。系统调用的实现原理包括用户程序发起系统调用、内核接收和处理系统调用、内核返回结果给用户程序等过程。

1. 用户程序发起系统调用：

用户程序通过汇编指令或 C 语言代码调用系统调用。在 RISC-V 架构下，系统调用使用特殊的汇编指令`ecall`来触发。用户程序在需要调用系统调用时，将系统调用号和参数传递给相应的寄存器，然后执行`ecall`指令。在 RISC-V 架构中，系统调用号存储在`a7`寄存器中，而系统调用的参数则存储在`a0`~`a6`寄存器中。`ecall`指令会导致 CPU 从用户模式切换到特权模式，进入内核态，转而执行操作系统内核的代码。

2. 内核接收和处理系统调用：

当用户程序发起系统调用后，CPU 进入内核态，操作系统内核接收并处理系统调用。在系统调用的处理代码位于`syscall`函数中。该函数首先获取系统调用号和参数，并根据系统

调用号调用相应的内核函数进行处理。在 xv6 中，每个系统调用对应一个特定的内核函数，例如读取文件的`sys_read`，创建进程的`sys_fork`等。内核函数执行系统调用所需的功能，如读写文件、创建进程、分配内存等。

xv6-riscv 中的系统调用处理过程涉及到保存和恢复进程的上下文（context switch）。在处理系统调用之前，操作系统会先保存当前进程的上下文，包括通用寄存器、程序计数器和其他特定寄存器。然后，根据系统调用号调用相应的内核函数进行处理，处理完毕后再将结果返回给用户程序。处理系统调用后，操作系统会恢复之前保存的进程上下文，将 CPU 从内核态切换回用户态，将控制权交还给用户程序。

3. 内核返回结果给用户程序：

内核处理完系统调用后，将返回值保存在指定的寄存器中，并将 CPU 从内核态切换回用户态，将控制权交还给用户程序。在 RISC-V 架构下，系统调用的返回值存储在`a0`寄存器中，用户程序可以通过查看该寄存器来获取系统调用的结果。

系统调用是实现用户程序与内核之间的交互的重要手段。用户程序通过系统调用请求操作系统提供服务，而内核通过处理系统调用来满足用户程序的请求。系统调用的设计和实现在操作系统中是一个复杂且关键的部分，涉及到系统调用的接口定义、参数传递、内核函数实现等方面。

内存管理：

1. 内存分页：

内存分页是指将物理内存划分成固定大小的页框（page frame），通常是 4KB 大小的页面。物理内存的地址空间被划分为一系列大小相等的页框，每个页框可以容纳一个页的数据。内存分页的目的是将物理内存划分成固定大小的块，方便管理和分配。在 RISC-V 架构下，内存分页的启用和禁用通过设置`satp`寄存器中的标志位实现，当启用分页时，处理器将使用页表进行虚拟内存映射，否则将直接使用物理地址。

2. 虚拟内存映射：

这个系统实现了虚拟内存映射，将每个进程的虚拟地址空间映射到物理内存中。每个进程都有自己的页表（page table），用于将虚拟地址映射到物理地址。当一个进程访问虚拟地址时，处理器会使用页表将虚拟地址转换为对应的物理地址。如果该虚拟地址没有映射到物理地址，处理器会触发缺页异常（page fault），操作系统将负责处理该异常，通常包括分配一个新的物理页框，然后更新页表以建立虚拟地址到物理地址的映射关系。

系统采用了两级页表（two-level page table）的方式进行虚拟内存映射。页表包括两个层级：页目录表（Page Directory Table, PDT）和页表（Page Table, PT）。页目录表中的每个条目指向一个页表，而页表中的每个条目指向一个页框。使用两级页表可以节省内存空间，因为每个页表条目只需要占用一个字节，而不是一个页面。此外，两级页表还能提高查找速度，因为只有两次查找就可以找到虚拟地址对应的物理页框。

3. 页面置换：

系统采用最简单的页面置换算法——FIFO（First-In-First-Out）。以支持更多进程的运行。当物理内存不足时，操作系统会选择最早进入物理内存的页面进行置换，即最先进入物理内存的页面将被替换出去。这样简单的页面置换算法能够较为高效地处理页面置换的问题，但是可能导致较大的内存碎片和性能问题。

内存管理部分实现了内存分页、虚拟内存映射和页面置换等功能。通过内存分页和虚拟内存映射，实现了进程间的隔离和内存保护，每个进程都有独立的虚拟地址空间。而页面置换机制则确保了操作系统可以支持更多进程的运行。

SD 驱动:

在 xv6 中, SD 卡是一种常见的存储设备, 用于在操作系统中进行数据读写操作。本文将介绍 xv6-riscv 中 SD 卡的驱动实现原理。

1. SD 卡简介

SD 卡 (Secure Digital Card) 是一种常用的闪存存储卡, 广泛应用于移动设备、数码相机、嵌入式系统等。SD 卡提供了高速、可靠的数据存储和传输功能, 使其成为 xv6-riscv 操作系统的重要外部设备之一。

2. SD 驱动硬件接口

在 xv6-riscv 中, SD 卡的驱动实现依赖于硬件接口的初始化和配置。SD 卡通常使用 SPI (Serial Peripheral Interface) 协议进行通信, 因此, 驱动程序需要通过 GPIO (通用输入输出) 端口和 SPI 控制器来与 SD 卡进行交互。

2.1 硬件初始化

驱动程序在系统启动时会进行硬件初始化, 以确保 SD 卡能够正确工作。初始化过程包括以下步骤:

配置 SPI 控制器: 驱动程序会设置 SPI 控制器的参数, 包括时钟频率、传输模式、数据位数等。

初始化 GPIO: 驱动程序会配置相应的 GPIO 引脚, 用于 SD 卡的片选 (Chip Select)、时钟、数据输入输出等控制。

2.2 SD 卡识别

在硬件初始化完成后, 驱动程序会进行 SD 卡的识别过程, 以确定 SD 卡的类型和容量。
发送初始化命令: 驱动程序会向 SD 卡发送初始化命令 (CMD0), 确保 SD 卡处于 IDLE 状态。

获取卡信息: 驱动程序会发送 CMD8 命令, 获取 SD 卡的电压范围和版本信息。

初始化 SD 卡: 驱动程序会发送 ACMD41 命令, 用于初始化 SD 卡, 并等待 SD 卡进入 READY 状态。

识别 SD 卡: 根据 SD 卡的响应, 驱动程序可以判断 SD 卡的类型 (SDSC、SDHC、SDXC 等) 和容量。

3. SD 驱动读写操作

驱动程序实现了读取和写入 SD 卡的功能, 使得 xv6-riscv 操作系统可以与 SD 卡进行数据的读写。

3.1 读取数据

当应用程序需要从 SD 卡读取数据时, 会通过系统调用 (例如 read) 来请求操作系统执行读取操作。驱动程序的读取操作涉及以下步骤:

发送读命令: 驱动程序会向 SD 卡发送读命令 (CMD17 或 CMD18), 指定要读取的数据块的地址。

接收数据: SD 卡接收到读命令后, 会返回数据块的内容。驱动程序通过 SPI 协议接收数据, 并将其存储在一个缓冲区中。

数据传输: 驱动程序将读取的数据块从缓冲区传输给应用程序, 完成数据读取过程。

3.2 写入数据

当应用程序需要向 SD 卡写入数据时, 会通过系统调用 (例如 write) 来请求操作系统执行写入操作。驱动程序的写入操作涉及以下步骤:

发送写命令: 驱动程序会向 SD 卡发送写命令 (CMD24 或 CMD25), 指定要写入的数据块的地址。

准备数据: 应用程序将要写入的数据存储在一个缓冲区中。

数据传输：驱动程序通过 SPI 协议将缓冲区中的数据块发送给 SD 卡，完成数据写入过程。

4. SD 驱动错误处理

在 SD 卡的读写过程中，可能会出现错误，例如 SD 卡未响应、数据传输错误等。驱动程序需要对这些错误进行处理，以确保 SD 卡的稳定工作。

错误检测：驱动程序会对 SD 卡的响应进行检测，判断是否发生了错误。

错误处理：当发现错误时，驱动程序会采取相应的措施，例如重新初始化 SD 卡、重新尝试读写操作等。

外部中断:

在 xv6 中，外部中断是实现操作系统与外部设备交互的重要机制之一，本文将介绍 xv6-riscv 是如何实现外部中断的。

1. 外部中断简介

外部中断是计算机系统中的一种机制，允许外部设备向处理器发出信号，以通知其发生了某种事件。在操作系统中，外部中断常用于处理硬件设备的输入输出、时钟中断、网络中断等。对于 RISC-V 架构的处理器，外部中断被设计为包含两种类型：时钟中断和外部设备中断。

2. xv6-riscv 外部中断的实现

在 xv6-riscv 中，外部中断的实现涉及到中断控制器（Interrupt Controller）和中断处理例程（Interrupt Handler）。具体实现过程如下：

2.1 中断控制器

xv6-riscv 采用的处理器一般会配备一个称为 CLINT（Core Local Interruptor）的本地中断控制器。CLINT 是 RISC-V 处理器的标准组件之一，负责处理与处理器核心相关的定时器中断和软件中断。在 CLINT 中，有一个时钟中断定时器，它以固定的频率产生时钟中断信号。此外，CLINT 还可以接收其他外部设备的中断信号。

2.2 中断处理例程

中断处理例程是用于处理中断的特殊函数，当外部中断发生时，处理器会暂停当前的执行流程，跳转到相应的中断处理例程中执行。在 xv6-riscv 中，处理器的中断处理例程被称为“trap”。

2.2.1 时钟中断处理

时钟中断是操作系统中常见的一种中断，它通常以固定的频率触发，用于维护系统时间以及进行进程调度。在 xv6-riscv 中，时钟中断由 CLINT 产生，当时钟中断发生时，CLINT 会向处理器发送一个时钟中断信号，处理器接收到该信号后会立即执行时钟中断的处理例程。在时钟中断处理例程中，操作系统会更新系统时间，调度器检查当前运行的进程是否已经用完了时间片（时间片轮转调度算法），如果用完则进行进程切换，将当前进程挂起，切换到下一个要执行的进程。

2.2.2 外部设备中断处理

除了时钟中断，外部设备中断也是 xv6-riscv 中的重要部分。外部设备中断可以来自外部硬件设备（如磁盘控制器、网络控制器等），当这些设备完成特定操作或者出现异常情况时，会向处理器发送外部设备中断信号。

在外部设备中断处理例程中，操作系统会处理特定设备的中断请求，可能涉及读写设备数据、更新设备状态、通知相关进程等。

2.3 中断处理过程

当处理器接收到中断信号时，会暂停当前的执行流程，保存当前的上下文信息（寄存器

内容等), 并跳转到相应的中断处理例程中执行。中断处理例程完成后, 处理器会恢复之前保存的上下文信息, 继续执行被中断的程序。

问题:

1. mount 权限不够问题

`docker run --privileged --name 自己的名字 -d 容器的名字`

```
error: unexpected argument 'proc-macro' found
Usage: cargo update [OPTIONS]
• For more information, try '--help'.
root@dc065e7c2f4:/rustsbi-qemu# rustup default stable
info: using existing install for 'stable-x86_64-unknown-linux-gnu'
info: default toolchain set to 'stable-x86_64-unknown-linux-gnu'

stable-x86_64-unknown-linux-gnu unchanged - rustc 1.71.0 (8ede3aa2 2023-07-12)

info: note that the toolchain 'nightly-x86_64-unknown-linux-gnu' is currently in use (overridden by '/root/rustsbi-qemu/rust-toolchain.toml')
root@dc065e7c2f4:/rustsbi-qemu# cargo test
warning: some crates are on edition 2021 which defaults to 'resolver = "2"', but virtual workspaces default to 'resolver = "1"'
note: to keep the current resolver, specify 'workspace.resolver = "1"' in the workspace root's manifest
note: to use the edition 2021 resolver, specify 'workspace.resolver = "2"' in the workspace root's manifest
   Compiling proc-macro2 v1.0.50
   Compiling io-lifetimes v1.0.4
error[E0635]: unknown feature 'proc_macro_span_shrink'
  --> /root/.cargo/registry/src/mirrors.ustc.edu.cn-61ef6e0cd06fb9b8/proc-macro2-1.0.50/src/lib.rs:92:30
92 |     feature(proc_macro_span, proc_macro_span_shrink)
   |                                ~~~~~
   |
   = help: to see what features are available, use `cargo tree --features`
   Compiling rustix v0.36.7
For more information about this error, try `rustc --explain E0635`.
error: could not compile 'proc-macro2' (lib) due to previous error
warning: build failed, waiting for other jobs to finish...
root@dc065e7c2f4:/rustsbi-qemu# cargo update -p syn --precise 1.0.99; cargo update -p proc-macro2 --precise 1.0.43
warning: some crates are on edition 2021 which defaults to 'resolver = "2"', but virtual workspaces default to 'resolver = "1"'
note: to keep the current resolver, specify 'workspace.resolver = "1"' in the workspace root's manifest
note: to use the edition 2021 resolver, specify 'workspace.resolver = "2"' in the workspace root's manifest
Updating 'ustc' index
```

2.

方法: `cargo update -p syn --precise 1.0.99; cargo update -p proc-macro2 --precise 1.0.43`

3. 启动 QEMU 控制台, 启动后按下 `ctrl + a c`

修改命令行不能使用的的问题, 使用的方法是在内核和 `console.c` 中添加 `uart.c` 键盘的初始化的逻辑。

修改内核初始化, 重构 `main` 函数, 加入对核的判断。

4. 无法从 sd 卡中读取相应的程序:

重写 `init` 函数----`-initcode`

采用二进制文件读取的方式, 使用 `inittest.c` 文件写出 sd 卡用户程序的调用顺序, 然后编译生成 `.bin` 文件, 最后在内核中添加一个大数组, 读取 `.bin` 文件中的二进制代码, 作为内核启动的第一个用户程序调用。

5. 读取的时候的页面对齐问题: (目前还没有解决)

参考文献:

1. "Operating Systems: Three Easy Pieces" by Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau
2. "The RISC-V Reader: An Open Architecture Atlas" by David Patterson and Andrew Waterman
3. xv6 GitHub repository and the accompanying documentation (<https://github.com/mit-pdos/xv6-public>)
4. RISC-V Instruction Set Manual Volume I: User-Level ISA, 版本 2.2, 编者: Andrew Waterman, Krste Asanović 等, 2017 年。该文档详细描述了 RISC-V 指令集的架构和用法, 包括 `ecall` 指令的用法和系统调用相关内容。
5. xv6-riscv 官方文档: <https://pdos.csail.mit.edu/6.828/2020/xv6.html>。
6. RISC-V Instruction Set Manual Volume II: Privileged Architecture, 版本 1.11, 编者: Andrew

Waterman, Krste Asanović 等, 2019 年。

7. xv6-riscv 官方文档: <https://pdos.csail.mit.edu/6.828/2020/xv6.html>。
8. xv6 官方 GitHub 仓库: <https://github.com/mit-pdos/xv6-public>
9. "Serial Peripheral Interface (SPI)" by Wikipedia contributors, Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/wiki/Serial_Peripheral_Interface
10. "Secure Digital" by Wikipedia contributors, Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/wiki/Secure_Digital
11. xv6 官方 GitHub 仓库: <https://github.com/mit-pdos/xv6-public>
12. RISC-V 官方网站: <https://riscv.org/>
13. "The RISC-V Instruction Set Manual Volume II: Privileged Architecture" by Andrew Waterman and Krste Asanović, Version 1.11, June 2019.