操作系统实验手册（2024年春季）

**实验要求：**

1. **本课程是实验课程，请每位同学确保在规定时间内完成实验。**
2. **本课程成绩由两部分构成，验收成绩占70%，实验报告成绩占30%。实验结束时，由老师上机验收成果，同学们提交实验报告。**
3. **本课程设置了两类实验：操作系统模拟实验和操作系统内核实验，模拟实验难度低于内核实验，同学们可以根据自己的情况进行选择。评分时会考虑难度系数，但完成内核实验的同学的得分并不一定就高于完成模拟实验的同学，还是要看大家的完成情况。**
4. **实验报告提交word格式或PDF格式文档，实验报告撰写要求参见“操作系统实验报告撰写要求”文档。用word编辑实验报告的同学请在文档结构图中保留目录结构。所有实验内容均写在一个实验报告中，每个实验报告文档需要有摘要、目录和参考文献，还需要记录在本次实验中遇到的问题及解决方案。实验报告不是写的内容越多越好，请用自己理解后的语言来描述，报告长度控制在50页以内。请保留封面的成绩评分表。**
5. **提交方式：请大家按指导老师要求的方式提交报告。实验报告的命名方式为：“学号-姓名-操作系统\*\*实验报告”，其中\*\*指的是模拟或内核。请在截止时间前提交。**
6. **以下情况会被额外扣分：**

* **没有在报告封面保留成绩评分表**
* **没有按要求命名实验报告文件**
* **实验报告内容不完整，格式不符合要求**

# 操作系统模拟实验

实验一：进程调度实验

【实验目的】

1. 理解在单处理器环境下的进程调度及状态转换的基本原理和关键点。
2. 编程实现时间片轮转调度、短进程优先调度和多级反馈队列调度算法。
3. 分析并比较不同调度算法的效率和公平性。

【实验要求】

1. 模拟每一个进程，需要包含到达时间，运行时间，优先级等属性。
2. 每一种调度算法都需要实现一个模拟函数，对进程结束时间进行仿真模拟。多级反馈队列调度算法至少包含三个队列，每个队列的时间片大小不同。
3. 对给定的一组进程，当采用不同的调度算法时，需仿真模拟这组进程的状态转换过程。进程的状态至少有就绪、运行和阻塞三种状态。
4. 当这组进程调度执行完成时，根据仿真模拟的结果，计算平均周转时间/平均带权周转时间，平均等待时间、平均响应时间、利用率等性能指标及公平性指标。
5. 能够提供合适的测试数据集，对每一种算法进行测试，并分析其结果。

【涉及的基本原理】

进程调度是指操作系统中关于处理系统进程运行顺序和方式的机制。在设计进程调度程序时，需要了解以下基本原理：

1. 进程的概念：进程是程序的一次执行过程，是系统进行资源分配和调度的基本单位。它由程序代码和相关数据、堆栈、PCB(Process Control Block, 进程控制块)等部分组成。
2. 进程的状态：进程最基本的三种状态是：就绪状态（Ready）、执行状态（Running）、和等待状态（Waiting）。在进程的生命周期中，会反复在这几种状态之间转换。
3. 调度器：负责根据特定策略，决定哪个进程占有CPU。调度器分为长程调度器（作业调度器）、短程调度器（CPU调度器）、中程调度器。本模拟实验仅涉及短程调度。
4. 进程调度策略：调度策略主要有先来先服务（FCFS）、最短作业优先（SJF）、优先级调度、轮转法（RR）、多级反馈队列等。这些调度策略均有其特定的场合和使用环境。
5. 多级反馈队列调度算法是一种兼顾了高响应比优先、短作业优先和时间片轮转等算法优点的算法。它将就绪队列分为若干个队列，每个队列的优先级不同；同时，各队列可以采用不同的调度算法，以达到最大程度地提高系统的吞吐量、响应时间以及带宽等性能指标。
6. 上下文切换：当CPU从一个进程切换到另一个进程时，需要保存当前进程的状态，并加载新的进程状态，这个过程被称为上下文切换，它会产生一定的开销。
7. 并发性：操作系统可以让多个进程“同时”运行，实际上是通过迅速地在不同进程间进行切换，为每个进程分配一段时间来执行，从而实现宏观上的并发。

【实验提示】

1. 在模拟实验中，进程的到达时间、运行时间以及进程的特性（CPU繁忙型、I/O繁忙型）通常是通过人工指定或使用随机数生成方法产生的。人工指定的方法更适用于测试环境，因为你可以控制所有的变量，使测试结果可预测并被理解。随机数生成的方法更能够创建更真实的运行环境，但在使用随机数生成方法确定进程到达时间和运行时间时，需要注意一些细节。例如，进程到达时间必须是递增的；运行时间应确保在合理范围内，不能太长也不能为0。同时，如果用随机方法生成的测试数据，务必记住保存这批数据或者保存随机种子，以确保实验结果的可重复性。
2. 在编程过程中，需要注意进程控制块PCB结构的选择，以便适应于不同算法的需求。
3. 在分析调度效果时，需要理解在不同的进程运行情况下（比如到达时间，服务时间，进程的数量等），不同的调度算法可能会有完全不同的表现。在进行平均周转时间等性能指标的计算时，也需要注意公平性的考量。
4. 在设计进程和调度算法时，要考虑清楚测试用例是否可以覆盖多种可能的情况，以便于发现并处理边缘情况。有时甚至要创建一些额外的测试用例去覆盖边缘情况，比如所有进程几乎同时到达，或者所有进程运行时间都一样等。

【验收标准】

1. 基本功能的实现：能够正确模拟时间片轮转调度、短进程优先调度和多级反馈队列调度算法，并且模拟的结果正确。
2. 选择合理的进程数据集进行展示：测试数据集能够为每种调度算法提供足够的测试覆盖。
3. 能够选择合适的性能评估指标和公平性指标来分析和评价各种调度算法。
4. 系统的稳定性和错误处理：程序在运行过程中是否能稳定执行，是否能有效地处理错误的情况。
5. 用户友好度：输出信息要能够清晰反映调度过程中进程的状态变化。
6. 代码质量：高质量的代码表现在清晰的组织结构，良好的命名规范，适当的注释等。良好的代码质量可以提高代码的可读性和可维护性。
7. 实验报告：一个好的实验报告应该详细描述算法的设计与实现的关键点、充分展示测试情况，记录遇到的问题及解决方案等，并对不同的调度算法进行性能和公平性分析，并得出你的结论。

实验二：虚拟页式存储管理系统

【实验目的】

1. 理解虚拟内存和物理内存之间的关系及转化，在什么情况下会发生缺页中断。
2. 理解和掌握各种页面置换算法的工作原理，包括先进先出（FIFO）、最近最少使用（LRU）、时钟置换（CLOCK）和随机置换算法，并通过模拟实验观察它们的执行过程。
3. 通过比较不同页面置换算法下系统的性能，理解不同算法的优缺点以及在何种情况下更适用。
4. 学习如何评估系统性能。通过实验结果，理解如何通过缺页中断次数、平均访问时间等指标去评估系统性能，并能合理地解释实验结果。
5. 从实验中体验和理解内存访问的局部性原理，提高自己对操作系统虚拟内存管理的理解。

【实验要求】

1. 设计并实现一个模拟的简单虚拟页式存储管理系统。在该系统中，应能模拟以下功能：创建进程的页表，为进程页面分配内存块，实现逻辑地址向物理地址的转换。当发生缺页时，通过缺页中断处理调入页面，如果需要，可以进行页面置换。
2. 制定一套模拟数据，包括要访问的页面、需执行的操作等，并按照虚拟页式存储管理的原则进行处理，展示每一步的执行结果。
3. 能够模拟出页面置换的情况，例如FIFO、LRU、CLOCK和随机置换等算法，比较并分析每种算法下系统的性能。

【涉及的基本原理】

1. 虚拟内存技术：这是一种使得计算机系统看上去具有比实际RAM更大的内存的技巧。它使用了硬盘空间作为对RAM的扩充并将之作为应用程序的虚拟存储器。
2. 页（Page）：操作系统管理内存的一个单位。操作系统会把逻辑地址空间分割为一系列连续的、固定大小的区域，这些区域就称为页。
3. 帧（Frame）：操作系统管理对应实际内存的一个单位，与页的大小一致。
4. 页表：虚拟页面和物理内存帧之间的映射存放在页表中。为了提高地址转换的效率，操作系统通常会采取一些策略来优化页表，如多级页表、倒置页表、TLB（Translation Lookaside Buffer）等。
5. 地址映射：操作系统会维护一个页表，记录页号和帧号的映射关系。当CPU发出一个虚拟地址时，操作系统会通过页表将虚拟地址转换为实际的物理地址。
6. 页面置换：由于内存有限，当分配给一个进程的所有页面都在使用中且需要加载新的页面时，操作系统必须选择一个页面将其移出内存以空出空间给新的页面，这个决定过程就是页面置换。常见的页面置换算法包括最近最久未使用（LRU）、最佳置换算法（OPT）、先进先出（FIFO）、时钟置换（CLOCK）等。
7. 缺页中断：当程序引用的页不在内存中时，会产生缺页中断，操作系统需要从磁盘中将页调入内存，并更新页表。

【实验提示】

1. 设计进程类，包含该进程的页表，页表中应包含页号和对应的物理块号等。
2. 设计内存类，模拟内存中的物理块，能够根据进程请求分配和回收物理块。
3. 设计地址转换函数，输入逻辑地址，输出物理地址。
4. 实现页面置换算法，对内存中的页面进行动态调整。
5. 缺页中断通常是因为进程尝试访问其虚拟内存空间中一个不存在于物理内存（RAM）中的页面时触发的，如果分配给该进程的物理内存已满，则需要进行页面置换。
6. 为了展示该内存管理系统，需设计合理的模拟数据，当运行内存管理程序时，展示每一步的内存分配、地址转换和页面置换情况。
7. 在模拟不同页面置换算法时，请选择合适的测试用例来体现内存访问的局部性原理，比如循环和递归时会访问重复的页面集，而访问数组或其它大型数据结构时会访问一大块连续的内存页面，这些情况在测试时需要考虑。
8. 分析并比较不同页面置换算法下系统的性能，需要定义一些性能指标，比如缺页中断次数、平均访问时间等。

【验收标准】

1. 基本功能的实现：能正确执行虚拟地址向物理地址的转换，正确实现了四种页面置换算法，包括LRU（最近最久未使用）、FIFO（先入先出）、CLOCK（时钟算法）和随机置换算法，能够在内存空间不足时，正确选择被替换的页面。
2. 选择合理的测试数据进行展示：能够设计合理的模拟数据，展示虚拟内存管理过程，能够通过模拟数据触发缺页置换过程，而不是直接指定并执行页面置换程序。
3. 系统的稳定性和错误处理：程序在运行过程中是否能稳定执行，是否能有效地处理错误的情况（如页面错误、地址越界错误）。
4. 用户友好度：系统应当对用户友好，易于使用。具体表现在接口设计的合理性，输出信息的清晰性等。
5. 代码质量：高质量的代码表现在清晰的组织结构，良好的命名规范，适当的注释等。良好的代码质量可以提高代码的可读性和可维护性。
6. 实验报告：一个好的实验报告应该详细描述系统的设计、实现的功能及使用方法，测试情况，以及遇到的问题、解决方式等。在实验报告中需要包含采用不同页面置换算法时系统性能的分析比较，需要采用合理的性能指标。

实验三：文件系统

【实验目的】

1. 理解文件系统的主要组成和工作原理。
2. 学习并掌握文件的存储方式，文件的创建、读写、删除等操作的实现机制。
3. 提高对文件系统内部组织和管理的认识及熟悉实际系统使用上的细节。

【实验要求】

1. 设计并实现一个可以支持树状目录的文件系统。
2. 能够执行基本的文件操作，如创建、打开、关闭、读取、写入、删除文件，创建、删除和列目录等。
3. 通过终端接口与用户交互，实现对模拟文件系统的操作。
4. 不能使用与文件操作相关的系统调用来完成该模拟文件系统的设计，需要基于自定义的文件、目录等数据结构来实现对文件、目录的管理。

【涉及的基本原理】

1. 文件系统的基本构造和功能：文件系统由文件和目录组成，带有访问权限和管理信息，实现了对底层存储设备的抽象。文件系统是操作系统用于存储、组织、共享和访问数据的一种机制。
2. 文件的存储方式：包括连续存储方式、链接存储方式和索引存储方式。
3. 文件和目录：文件系统将所有的信息存储在文件中。每个文件都有一个唯一的名字，并放置在目录中。目录本身也是文件，它包含一组链接，每个链接将一个文件名与一个文件相关联。
4. 权限和特权：文件系统为每个文件和目录都定义了一组权限，如读（r）、写（w）、执行（x）等。还可以定义谁可以使用这些权限，如文件所有者，所在组的成员，或者其他人。
5. 空间管理：文件系统管理存储设备上的可用空间。当文件被创建或扩展时，文件系统需要找到可用的空间。当文件被删除或缩小时，其空间需要被回收。
6. 文件系统的层次结构：文件系统通常使用一个树形结构来组织文件和目录。树的根是一个特殊的目录，称为根目录。每个目录可以包含文件和其他目录。
7. 硬链接和软链接：硬链接是另一个指向同一个文件的引用，而软链接（或符号链接）则是指向另一个链接的引用。使用链接可以方便地共享和组织文件。
8. 一致性和完整性：为了保证在系统崩溃或断电情况下数据的一致性和完整性，文件系统需要在修改数据时使用某种形式的原子操作或者日志。当操作失败时，可以回到操作前的状态。

【实验提示】

1. 首先确定你的文件系统的基本结构和组织方式，需要思考，文件采用什么存储方式才能更有效地既支持大文件，又支持小文件，空闲空间如何组织和管理，目录如何组织才能支持多级目录。设计合理的文件和目录的数据结构。
2. 实现文件系统的初始化函数，可以设定一个大小固定的 Block 来模拟磁盘空间，初始化时所有 Block 为空闲。
3. 编写创建文件的函数，该函数需能够为新文件分配 Block，更新文件的元数据metadata，将新文件添加到其所在目录下。
4. 编写读取和写入文件的函数，这两个函数需要通过文件的 metadata 找到文件在 Block 上的位置。在写入或删除文件时，需要考虑 Block 空间的分配和回收。
5. 编写删除文件和目录的函数，需要释放文件对应的 Block 并从所在目录中删除该文件。
6. 编写用户交互接口，可以通过终端命令来执行对文件的创建、打开、关闭、读取、写入、删除文件操作以及对目录的创建、删除、列目录等操作。在实现用户接口时，需要对输入命令进行充分的验证和错误处理。

【验收标准】

1. 功能完整性：一个好的模拟文件系统应该实现操作系统中文件系统管理的相关基础功能，这些包括文件的创建、打开、关闭、读取、写入、删除，目录的创建、删除，以及权限管理等。

2. 文件系统的组织和管理：模拟文件系统应刻画出现实文件系统中的数据组织方式。例如，如何组织文件存放、如何管理空间分配、如何划分目录等，都是评价的重要考量。不能使用操作系统提供的文件、目录相关系统调用来完成对文件和目录的操作。

3. 代码质量：代码质量是评判所设计文件系统的一项重要标准。好的代码应该简洁，清晰，易于利用和理解。应该有适当的注释，良好的命名规范，以及恰当的异常处理。

4. 错误处理能力：模拟文件系统应具备良好的错误处理能力。例如，对于非法输入应给出明确的错误提示，处于故障状态时应提供合理的恢复方案。

5. 用户界面：虽然这是一个模拟的系统，但是一个直观，简单，易用的用户界面也是评价一个文件系统好坏的一个标准。系统应该提供清晰的指导，易于理解的命令和明确的反馈。

6. 实验报告：一个好的实验报告应该详细描述系统的设计、实现的功能及使用方法，测试情况，以及遇到的问题、解决方式等。

# 操作系统内核实验

一、预备知识——工具及使用方法简介

以下工具是分析和修改Linux内核代码的基础，掌握它们将大大提高工作效率。建议在实践中多尝试，以充分掌握其强大的功能。

1. Visual Studio Code (VSCode)

作用

* VSCode 是一款流行的代码编辑器，支持多种编程语言，包括C/C++（Linux内核主要使用语言）。它提供了代码高亮、智能提示、代码调试等功能。

基本使用方法

* 安装插件：安装C/C++插件以支持相关语言的代码高亮和错误提示。
* 打开项目：通过“文件”菜单打开Linux内核代码所在的文件夹。
* 编辑与浏览：利用代码高亮和文件导航功能，浏览和编辑代码。

参考链接

* [VSCode 官方网站](https://code.visualstudio.com/)

VSCode插件

以下是一些有用的VSCode插件，可以帮助您在阅读Linux源码时定位函数和变量：

(1). C/C++

* 用途: 提供C/C++语言的智能提示、代码导航、符号搜索等功能。
* 特点: 支持智能感知 (IntelliSense)、调试和代码重构。

(2). GitLens

* 用途: 增强VSCode的Git功能，可以轻松浏览和理解代码库的历史。
* 特点: 查看文件历史、比较提交差异、查看谁更改了特定代码行等。

(3). Code Spell Checker

* 用途: 检查代码中的拼写错误，对于阅读和理解源码尤其有帮助。
* 特点: 支持多种语言，可以帮助识别和更正代码中的拼写错误。

(4). ctags

* 用途: 跟踪代码中的符号，如函数、变量等，帮助开发者在大型代码库中快速查找特定的函数定义、变量声明等。
* 特点: 快速导航到源码中的特定符号。

(5). Include Autocomplete

* 用途: 自动补全C/C++的头文件。
* 特点: 在编写或查看代码时自动建议相关的头文件。

(6). Sourcegraph

* 用途: 为代码提供强大的搜索和跨仓库的代码智能。
* 特点: 快速查找和理解开源代码库中的代码。

2. Git

作用

* Git 是一个版本控制系统，用于跟踪Linux内核代码的历史更改，方便协作和版本管理。

基本使用方法

* 克隆仓库：使用 git clone 命令克隆Linux内核的代码库。
* 查看历史：使用 git log 查看代码的修改历史。
* 分支管理：使用 git branch 和 git checkout 管理和切换分支。

参考链接

* [Git 官方文档](https://git-scm.com/doc)

3. GDB

作用

* GDB 是GNU Debugger，用于调试程序，特别是C/C++编写的程序。

基本使用方法

* 启动GDB：在命令行中使用 gdb [程序名] 启动GDB。
* 设置断点：使用 break [函数名或行号] 设置断点。
* 运行与调试：使用 run 开始运行程序，遇到断点时会停下来，可用 next、step 等命令逐行调试。

参考链接

* [GDB 官方文档](https://www.gnu.org/software/gdb/documentation/)

4. QEMU

作用

* QEMU 是一个通用开源的模拟器，可以在任何支持的架构上运行任何机器的操作系统，因此可以用来运行和调试Linux内核。

基本使用方法

* 安装QEMU：根据操作系统安装QEMU。
* 启动内核：使用QEMU启动编译好的Linux内核。
* 调试内核：与GDB结合使用，进行内核级别的调试。

参考链接

* [QEMU 官方网站](https://www.qemu.org/)
* [知乎专栏：windows下QEMU的安装和使用](https://zhuanlan.zhihu.com/p/670696946?locale=en)
* [知乎专栏：QEMU安装和基本使用方法](https://zhuanlan.zhihu.com/p/479666128?locale=en)

5. Diff

作用

* Diff 是一个命令行工具，用于比较文件的不同之处。在分析Linux内核代码时，它可以帮助你识别两个版本之间的差异。

基本使用方法

* 比较文件：使用命令 diff 文件1 文件2 来比较两个文件的内容差异。
* 输出差异：Diff 会列出两个文件不同的行，并显示如何从一个文件修改到另一个文件。
* 选项：Diff 提供多种选项来调整比较方式，如 -u（统一格式显示差异）。

参考链接

* [Diff 官方文档](https://www.gnu.org/software/diffutils/manual/diffutils.html)

6. Meld

作用

* Meld 是一个图形界面的文件比较工具，支持两个或三个文件的比较。它非常适合于可视化地比较和合并代码变更。

基本使用方法

* 安装Meld：根据操作系统安装Meld。
* 启动Meld：在图形界面中打开Meld。
* 选择文件进行比较：选择你想要比较的文件或目录。
* 查看和合并差异：Meld 显示不同文件的差异，并允许你编辑文件和合并变更。

参考链接

* [Meld 官方网站](http://meldmerge.org/)

二、实验环境的搭建

在x86架构的机器上构建基于RISC-V的xv6实验环境需要使用交叉编译技术和模拟器来模拟RISC-V环境。在搭建实验环境之前，大家需要了解，什么是交叉编译？为什么需要交叉编译工具链？以及如何在x86架构的机器上构建支持RISC-V架构的操作系统xv6运行的环境？以下是构建实验环境的步骤：

1. 安装RISC-V编译器和工具链

* 安装能够编译RISC-V目标代码的交叉编译器。可以通过官方RISC-V GNU工具链源代码进行编译安装或者使用已经构建好的工具链。如果你使用的是Linux系统，可以通过包管理器来安装他们。例如，在Ubuntu系统中，使用以下命令来安装：

sudo apt-get install gcc-riscv64-linux-gnu

* 如果构建好的RISC-V编译器和工具链在你的系统中不可用，那么就需要下载源代码来编译安装RISC-V编译器和工具链。

1. 获取xv6源代码

* 从MIT 6.828课程网站或者官方GitHub仓库获得xv6的源代码。
* 基于RISCV版本的xv6：https://github.com/mit-pdos/xv6-riscv

1. 安装QEMU模拟器

* QEMU是一个通用的开源机器模拟器和虚拟化器。请确保安装了支持RISC-V的QEMU版本。
* 一些Linux发行版可以直接通过包管理器来安装适当版本的QEMU。比如：

sudo apt-get install qemu-system-misc

1. 编译xv6源代码

* 在xv6源代码目录下，通常有一个Makefile文件定义了如何构建系统。使用make命令及RISC-V工具链构建xv6，命令类似于：

make ARCH=riscv CROSS\_COMPILE=riscv64-linux-gnu-

* 编译完成后，得到适用于RISC-V架构的xv6内核镜像。

1. 在QEMU中运行xv6

* 使用QEMU来启动xv6。运行类似于以下的命令来在模拟器中启动xv6操作系统，假设xv6内核位于"xv6-riscv/kernel"路径下：

qemu-system-riscv64 -machine virt -nographic -bios none -kernel xv6-riscv/kernel

* 在QEMU中启动xv6后则可通过控制台与之进行交互。

1. 测试和调试

* 在模拟环境中，运行xv6的用户程序，并进行开发、测试和调试。

以上是在x86的机器上构建RISC-V的xv6实验环境的主要步骤，涉及获取源代码、准备编译工具链、编译内核、以及在模拟器中运行和调试。请大家根据自己的系统配置和环境进行相应的调整和修改。如果上述步骤或命令有变化，建议查阅最新的官方xv6或RISC-V文档。

三、系统调用实验

实验任务：添加并测试自定义系统调用

在RISC-V体系结构上运行的xv6操作系统中引入一个新的系统调用，该调用执行一个简单任务，比如返回系统中当前进程的数量。

实验目的

1. 熟悉RISC-V体系结构。
2. 理解xv6中系统调用的工作原理。
3. 获得修改xv6源代码的经验。

实验步骤

1. 定义系统调用接口

在user/user.h头文件中创建一个新的系统调用函数原型。在本实验中，比如可能命名为int getprocs(void);。

1. 实现系统调用

在xv6内核中，创建一个新的C文件或在现有文件中来实现系统调用。该函数可能如下所示：

int sys\_getprocs(void)

{

return getprocs();

}

然后在内核中实现实际的getprocs()函数，该函数与进程表交互以计算活跃的进程数量。

1. 分配系统调用号

通过编辑kernel/syscall.h文件并添加一个新的define，为你的自定义系统调用分配一个唯一的系统调用号，例如：

#define SYS\_getprocs 22

1. 更新系统调用调度程序

修改kernel/syscall.c中的系统调用调度器函数，以处理新的系统调用编号。你还需要为sys\_getprocs函数添加一个原型。

extern int sys\_getprocs(void);

...

void syscall(void)

{

...

switch(myproc()->trapframe->a7){

...

case SYS\_getprocs:

myproc()->trapframe->a0 = sys\_getprocs();

break;

...

}

}

1. 用户空间库

编辑user/usys.pl脚本以定义新系统调用的用户空间映射，它将生成usys.S汇编代码。

添加类似的行：

entry("getprocs");

1. 测试系统调用

编写一个用户程序，调用getprocs()并将计数显示到控制台。

#include "kernel/types.h"

#include "user/user.h"

int main(int argc, char \*argv[])

{

int count = getprocs();

printf("There are %d active processes.\n", count);

exit(0);

}

在进行这些更改后，需要重新编译内核和用户程序。

注意事项：

1. 确保在xv6源码仓库中编辑正确的文件。
2. 保持对源代码的持续备份，以防工作丢失。
3. 增量测试以便及早发现错误。
4. 注意xv6中用于系统调用的约定，包括错误处理和返回值。

按照以上步骤，你应该能够在RISC-V的xv6中创建并测试一个新的系统调用。这是一个操作框架，你需要通过探索详细填写具体内容，比如计算进程数量的实际机制，这将涉及遍历xv6的进程表。

四、内存管理实验

实验任务：动态内存分配器的实现

设计并实现一个类似malloc的动态内存分配器，用于为用户态程序运行时分配和释放不同大小的内存块。

实验目的

1. 认识和实践xv6操作系统内存管理方法。
2. 通过设计和实现类似malloc的动态内存分配器，理解操作系统如何处理内存分配和回收，以及如何动态处理不同大小内存块的请求。

实验步骤

1. 研究当前xv6系统中内存管理的机制，特别是kalloc.c和vm.c中的代码，理解内核使用kalloc()和kfree()来分配和释放内存的机制。kalloc()仅用于xv6的内核空间，并且它总是分配固定大小的内存块(4KB)。
2. 动态内存分配策略可以借鉴经典的动态内存分配算法，比如伙伴系统或者slab分配器。设计合适的支持内存分配与合并的数据结构，比如空闲链表、伙伴系统等。并设计内存分配和释放的接口函数，比如malloc(size)和free(pointer)。
3. 修改kernel/kalloc.c文件，在xv6内核中实现这种内存分配策略：按照设计的接口，实现分配器。分配器应能够管理多种大小的内存块，并在内核中高效运行。
4. 编写测试用例来验证分配器在不同情况下的行为，包括但不限于内存申请、释放和边界情况处理，验证内存分配器的正确性、以及处理不同大小需求的能力。
5. 评估性能：分析和优化内存分配器的性能，关注内存使用效率和分配速度。
6. 文档和报告：完成详细的实验报告，介绍你理解的xv6的内存管理机制，说明你的动态内存分配器的设计思路、实现过程和遇到的挑战，以及如何解决这些挑战。

**注意事项：**

* 要理解xv6内核没有类似malloc的动态内存分配器的现状，且默认只使用4KB的内存块。
* 在不修改原有内核内存分配器的情况下，单独为用户空间程序设计一种可以分配小内存块的动态内存分配策略，这样做可以降低设计的复杂性。
* 用户空间的动态内存分配是在进程的虚地址空间的堆区（Heap）进行的。堆的初始大小是很小的，但是当应用程序通过malloc函数请求更多内存时，堆区能够扩展。在xv6中，堆区的扩展可以通过sbrk系统调用或者其他类似的机制实现。堆随着动态分配的内存不断增加而增长，通过sbrk可以减少堆的大小，但实际应用中很少这么做。在利用sbrk系统调用来扩展堆区时，有两种策略，一种是即时分配，即每次用户调用malloc请求内存时，就调用sbrk来准确扩大堆区以满足要求；另一种是批量分配，即提前使用sbrk系统调用将堆区扩大到一定的容量，创建一个内存池，然后按需从这个内存池中划分内存给malloc请求。这两种方法有各自的优缺点，大家需要在内存的有效利用率和性能开销之间进行平衡，从而选择一种合适的实现策略。
* 尽管xv6采用的是虚拟页式存储管理方法，但在设计malloc函数时，实际上是在进程的虚拟地址空间内部进行内存的分配与回收，而不是直接在物理内存上执行这些操作。malloc负责从虚拟地址空间的堆区域分配一段连续的地址给应用程序使用，这些虚拟地址随后会通过页表映射到物理地址上。操作系统负责确保足够的物理内存页面可用来支持这些虚拟地址的映射，或者在物理内存吃紧的时候使用诸如换出（paging out）等技术来管理内存压力。因此在实现动态内存分配时，只需维护虚拟地址空间中堆区的空闲和占用情况，当用户通过malloc申请内存时，要找到足够大的一块空闲虚拟内存来满足请求，并修改相应的内存管理数据结构，当释放内存时，重新将该虚拟内存区域标记为空闲状态。
* 考虑内存碎片问题，尤其是在长时间运行的系统中。尽可能合并相邻的空闲块来减少内存碎片。
* 保持内存对齐是很重要的，以避免潜在的效能问题。
* 确保所有分配的内存在不再使用时能够被正确释放，避免内存泄露。
* 在xv6中编写了malloc动态内存分配器之后，需要使用xv6的用户空间程序来编写测试用例，这就意味着你编写用户程序时所使用的函数和系统调用都应该是xv6所支持的。
* 测试用例可以考虑以下几个方面的测试：
  + 基本的内存分配和释放：确认内存可以被分配并被正确释放。
  + 边界条件测试：例如尝试分配非常大或者非常小的内存块，以及尝试释放一个未分配的内存块。
  + 随机大小和顺序：模仿实际应用中的内存分配模式，申请随机大小的内存块，并在随机时刻释放。
  + 压力测试：连续分配和释放内存，测试分配器在高负载下的表现。
  + 内存覆写测试：分配内存，并写入数据后，再分配新的内存块，确保前一个内存块的数据未被改变。
* 动态内存分配器的测试样例框架如下，大家可以在此基础上添加更多的测试用例：

#include "types.h"

#include "user.h"

#include "stat.h"

#define LARGE\_SIZE 4096

#define MAX\_ALLOC\_SIZE 1024

char\* test\_malloc(int size) {

// 此处用引用你实现的malloc函数

return (char\*)sbrk(size);

}

void test\_free(char \*ptr) {

// 显式释放你的malloc函数申请的内存

}

//用于测试内存是否可以正确地写入和验证数据

void test\_memory(char \*ptr, int size, char pattern) {

for(int i = 0; i < size; i++) {

ptr[i] = pattern;

}

for(int i = 0; i < size; i++) {

if(ptr[i] != pattern) {

printf(1, "Memory check failed at position %d\n", i);

exit();

}

}

}

int main(int argc, char \*argv[]) {

// 基本的申请和释放测试

char \*ptr = test\_malloc(100);

if(ptr == 0) {

printf(1, "test\_malloc failed to allocate memory\n");

exit();

}

test\_memory(ptr, 100, 0xAA);

test\_free(ptr);

// 其他测试添加在此处

printf(1, "All tests passed\n");

exit();

}

* 分配器需考虑线程安全问题，因为它会由多个内核线程在并发环境下使用。