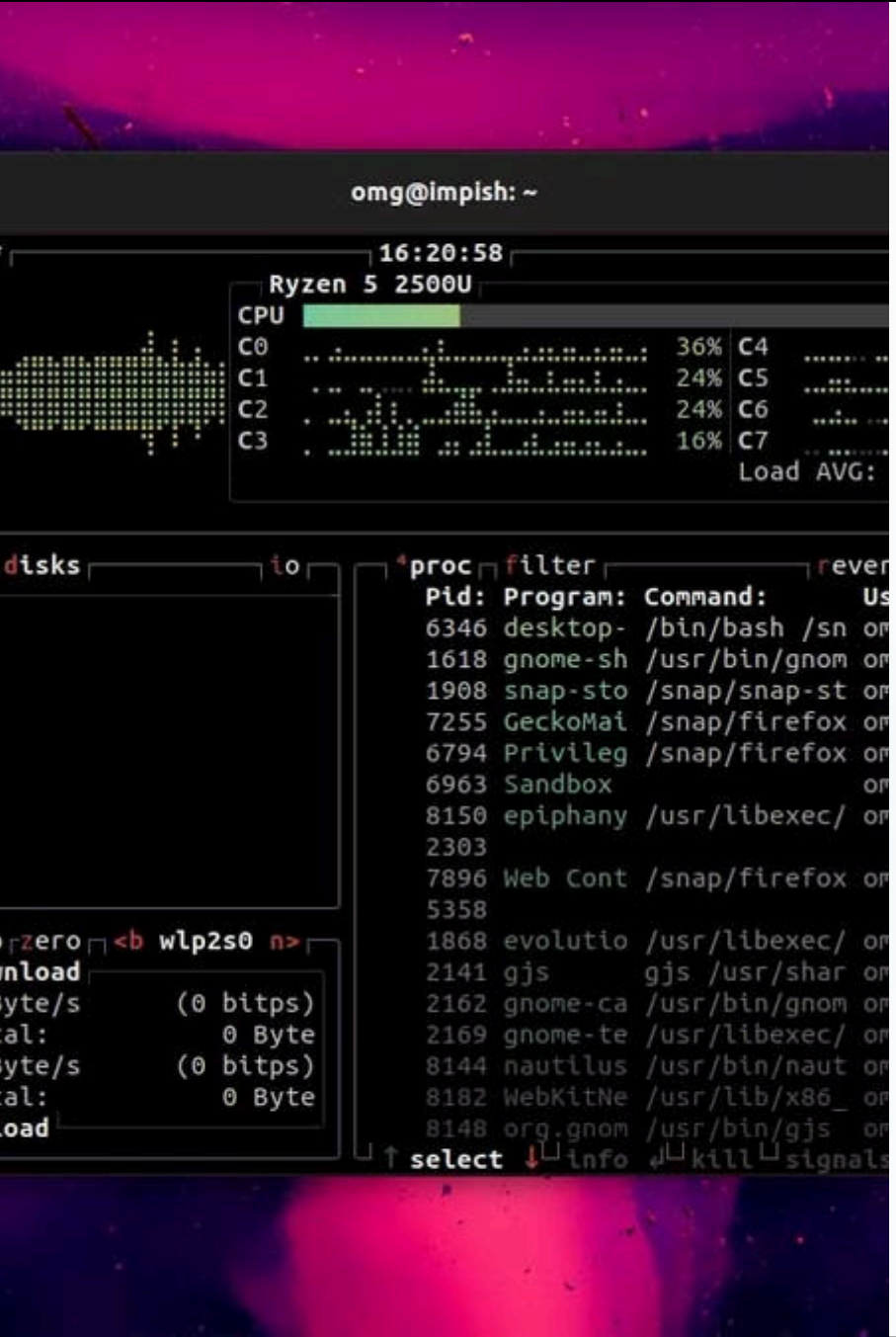


Linux基础知识

Linux操作系统以其强大的文件处理能力和灵活的管理特性在专业领域广受欢迎。

了解Linux文件操作和管理是掌握操作系统的重要一步，它包括文件的基本操作、权限设置、以及高级管理技巧。

操作系统是什么

操作系统是一个控制和管理计算机硬件与软件资源的软件系统。它提供了一个用户与计算机硬件之间进行交互的界面，并且负责分配硬件资源、管理文件系统、调度任务等。操作系统是计算机系统中最底层的软件，具有决定系统性能和稳定性的重要作用。

操作系统是什么

定义：操作系统（OS）是计算机系统中管理硬件与软件资源的系统软件。它为应用程序提供基础服务。

功能：

- **资源管理**：管理CPU、内存、磁盘、网络等硬件资源。
- **进程管理**：创建、调度、终止进程。
- **内存管理**：分配和回收内存。
- **文件管理**：提供文件系统，管理数据存储。
- **用户界面**：提供与用户交互的界面，如命令行和图形用户界面

什么是Linux

Linux是一种开源的操作系统，具有多样化的发行版本，如Ubuntu、Red Hat和CentOS等。它建立在UNIX操作系统的基础上，提供了稳定、可靠和安全的环境，适用于各种硬件平台和应用领域。

与Windows的区别：

- Linux是开源的
- Linux更加稳定和安全，广泛应用于服务器领域。
- Linux提供了更强大的开发工具和自定义选项。

与Mac的区别：

- Linux是开源的。
- Linux可以在各种硬件平台上运行，而Mac是专为苹果硬件开发的操作系统。
- Mac在用户界面和用户体验方面提供了独特的设计和功能。

Linux操作系统特点

开源性

Linux提供了一个开放源代码的环境，允许用户自由地访问、修改和再发布其源代码，促进了全球范围内的协作和创新。

免费使用

与许多操作系统收费不同，大多数Linux发行版可以免费获取，这降低了用户的入门门槛。

定制性强

用户可以轻松地定制自己的Linux系统，选择合适的桌面环境和必要的软件，以满足个人的特定需求。

高度安全

Linux被认为是最安全的操作系统之一，拥有严格的权限分级和优秀的访问控制机制。

为什么服务器采用Linux

Linux稳定可靠，支持高并发和大规模数据处理。

Linux具备强大的网络和服务器管理工具，方便灵活的配置和管理。

Linux开源特性受到企业青睐，可定制和优化，提高服务器性能和安全性。

使用Linux服务器可以节省成本，免费且无需高额许可费。

Linux特点

一切皆文件

在Linux中，无论是硬件设备、目录、常规文件还是网络套接字等资源，都被抽象为“文件”，并可通过统一的系统调用来操作。

强大的命令行工具

Linux提供了丰富的命令行工具，如bash shell、grep、sed、awk、find等，这些工具可以高效地处理文本、查找信息和管理系统。

模块化设计

Linux内核采用模块化设计，允许动态加载和卸载驱动程序、文件系统以及其他内核模块，使得系统可以根据需要灵活扩展功能。

多用户与多任务

支持多个用户同时操作，能够高效管理多个任务。

Linux文件操作与管理

文件系统

文件系统（File System）是操作系统中管理存储设备（如硬盘、SSD、USB 驱动器等）上文件的方式和结构。文件系统决定了如何组织、存储、访问和管理文件和数据。它提供了一个抽象层，使用户和应用程序可以方便地与文件和目录交互，而不需要关心底层硬件的细节。

文件系统的作用

存储与组织文件

文件系统负责将数据保存到存储设备中，并且对数据进行组织。它将存储设备划分为若干个文件和目录，并且提供一种访问文件的方式（如通过文件名或路径）。

数据管理与检索

文件系统通过文件名、路径等方式提供对存储在硬盘上的文件进行访问的途径。它为每个文件分配存储位置，并通过特定的数据结构（如 i-node）记录文件的元数据和存储位置。

访问控制与权限管理

文件系统可以提供文件的权限管理，控制用户对文件的访问权限（读、写、执行权限）。它通过文件的元数据（如文件的所有者、权限位等）来确保文件的安全性和访问控制。

数据完整性与安全

文件系统可以通过日志、校验等方式确保数据的一致性和完整性，尤其是在突然断电或系统崩溃的情况下，有些文件系统还支持数据加密功能。

文件系统的组成部分

文件

文件是文件系统中最基本的存储单位，用于保存用户和系统的数据。文件可以是文本文件、二进制文件、图像、视频等。

目录（文件夹）

目录是文件的容器，用来组织和管理文件。文件系统通过目录将文件进行层次化管理，使用户能够通过路径轻松访问文件。

元数据

每个文件和目录都有相应的元数据（如 i-node），包括文件的大小、权限、创建和修改时间、所有者等信息。这些元数据由文件系统管理，并提供给操作系统和用户使用。

路径

路径是文件和目录在文件系统中的位置，分为绝对路径和相对路径。路径帮助用户快速找到文件在存储设备中的具体位置

Linux的文件系统

在 Linux 系统中，**文件系统的数量是动态的**，具体数量取决于系统中**挂载了多少存储设备或分区**，以及每个存储设备使用的文件系统类型。因此，Linux 系统中可以有**多个文件系统**，每个文件系统可以是不同的类型，挂载在不同的目录下。我们可以通过挂载多个分区、设备、网络存储等方式来扩展文件系统的数量

一个 Linux 系统中可能有多少文件系统?

1. **根文件系统 /**:
   - 这个是系统启动时挂载的最基本的文件系统，它包含了操作系统的基本文件。
2. **挂载的分区**:
   - 系统可以将多个硬盘分区挂载到不同的目录。例如，将 /home 挂载到一个独立的分区，或者将 /var 挂载到另一个分区。
   - 你可以有多个文件系统，分别用于 /boot、/home、/var 等目录。
3. **外部设备**:
   - 外部 USB 驱动器、网络存储设备等也可以使用文件系统，它们通过挂载被 Linux 系统识别和访问。
   - 例如，挂载一个 exFAT 的 USB 驱动器到 /media/usb。
4. **伪文件系统**:
   - 例如 /proc、/sys、/dev 等并不存储用户文件，但它们作为特殊的文件系统存在于 Linux 中。

文件描述符

文件描述符的定义

在Linux中，文件描述符是一个整数，它代表了一个打开文件、设备或网络套接字的引用。

操作系统使用文件描述符来追踪每个进程所打开的文件和I/O资源。

标准文件描述符

0：标准输入（stdin），用于接收输入。

1：标准输出（stdout），用于输出信息。

2：标准错误（stderr），专门用于输出错误信息。

文件描述符

文件描述符是一个抽象指标，用于表示对文件或其他I/O资源的访问。

在Unix-like操作系统中，文件描述符是一个非负整数，用于标识已打开的文件、设备或网络套接字等资源的引用。

每个进程都有一张独立的文件描述符表，用于访问和管理其拥有的所有打开的I/O资源。

Linux内核为每个进程维护了一个文件描述符表，记录描述符关联的资源和状态信息。

通过系统调用操作文件描述符时，内核会根据描述符查找对应的资源并执行相应的操作。

文件描述符表

Linux内核文件管理

实际上，关于文件描述符，Linux内核维护了三个数据结构：

- 进程级的文件描述符表
- 系统级的打开文件描述符表
- 文件系统的i-node表

进程级的文件描述符表

- **概念**：每个进程都有一个**文件描述符表**，其中存放了当前进程所有打开文件的**文件描述符**。文件描述符是一个非负整数，作为标识符来表示进程打开的文件。
- **作用**：当进程调用 `open()`、`read()`、`write()` 等系统调用时，操作系统通过文件描述符找到该进程对应的文件。

例子：

```
int fd = open("file.txt", O_RDONLY); // 打开文件，fd 是文件描述符 read(fd, buffer, size); // 使用 fd 读取文件数据
```

关键点：

- 文件描述符是每个进程私有的，进程内部用来管理打开的文件。

系统级的打开文件表

- **概念**：系统级的打开文件表是 Linux 内核维护的一个表，用来存储系统中所有打开的文件的信息。无论是哪个进程打开的文件，这张表统一管理。
- **作用**：该表记录了文件的**读写偏移量**、**访问模式**（读、写等）、**引用计数**等。

系统级的打开文件表

例子：

- 当两个进程 A 和 B 同时打开同一个文件时，它们的文件描述符会指向**同一个系统级打开文件表**条目。
- 多个文件描述符可以共享同一个文件的读写位置（偏移量），因此它们的文件操作状态是共享的。

关键点：

- 系统级打开文件表允许多个进程共享对同一个文件的访问，而不会重复打开文件资源

文件系统的 i-node 表

- **概念**：i-node 表是文件系统层次的一个表，记录了文件的**元数据**，如文件大小、权限、创建时间等。每个文件和目录都有一个唯一的 i-node。
- **作用**：i-node 中还包含指向文件实际数据块的指针，这些指针告诉系统文件内容存放在哪里。

文件系统的 i-node 表

例子：

- 当进程打开文件时，内核通过文件的路径找到对应的 i-node，读取文件的权限、大小等信息，并找到文件的实际存储位置。

关键点：

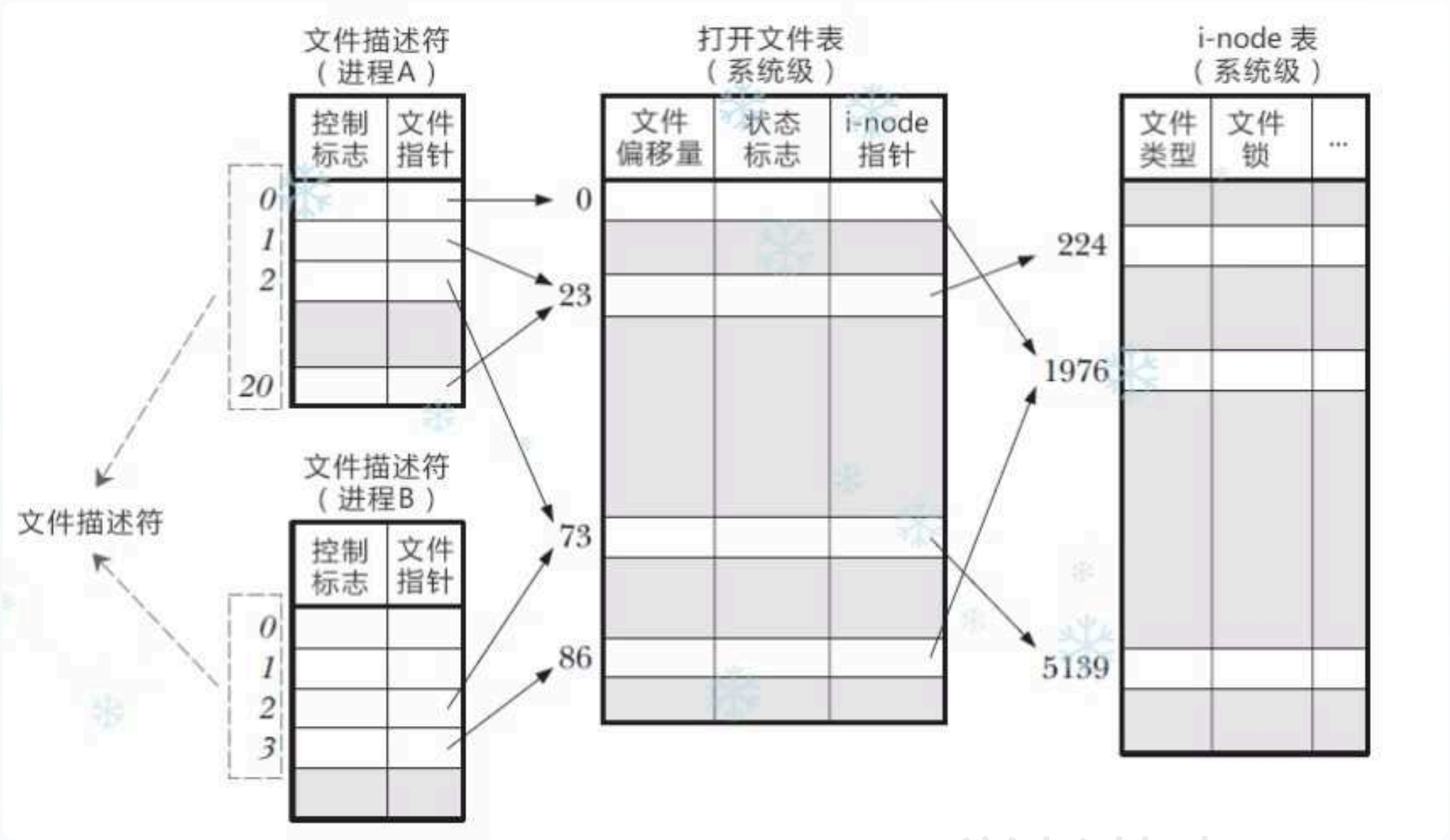
- i-node 负责管理文件的元数据信息，而文件的数据存放在磁盘块中，i-node 通过指针指向这些数据块

实际过程

文件的打开过程：

1. **进程级文件描述符表**：
   - 当进程调用 `open()` 打开文件时，操作系统会在该进程的文件描述符表中为其分配一个文件描述符（如 `fd = 3`），并指向系统级打开文件表中的相应条目。
2. **系统级打开文件表**：
   - 系统级打开文件表中存储该文件的状态信息（如读写偏移、打开模式），并通过一个指针指向该文件在 i-node 表中的条目。此时，系统会检查是否已有其他进程打开该文件，如果有，则会共享该条目。
3. **文件系统的 i-node 表**：
   - 系统级打开文件表条目中的指针指向该文件的 i-node。通过 i-node 表，操作系统可以读取文件的元数据（如文件大小、权限）和文件数据的存储位置。

实际过程



软链接与硬链接

在 Linux 文件系统中，**软链接（Symbolic Link）** 和 **硬链接（Hard Link）** 是两种用于创建文件引用的机制。它们允许多个文件名指向同一个文件，但在实现方式和行为上有很大的不同。

软链接（Symbolic Link）

软链接，又称为**符号链接**，是一个指向另一个文件或目录的**引用**，类似于 Windows 系统中的快捷方式。

特点：

1. **软链接是一个独立的文件**，存储的是另一个文件或目录的路径。
2. 软链接和目标文件是分开的，**它们拥有不同的 i-node**。
3. 如果目标文件被删除或移动，软链接会**失效**，因为它只是一个指向路径的引用，称为“悬空链接”。
4. 软链接可以指向**文件或目录**，甚至可以跨越不同的文件系统。

软链接的使用场景：

- 软链接常用于创建快捷方式，或者在不同目录下引用相同的文件。
- 软链接可以方便地创建指向文件或目录的引用，而不影响原始文件的物理位置。

软链接的优缺点：

- **优点**：
  - 可以跨文件系统创建链接。
  - 可以指向目录。
- **缺点**：
  - 目标文件被删除后，软链接就会失效，无法正常访问。
  - 软链接占用少量额外的存储空间，因为它存储的是指向目标的路径。

硬链接（Hard Link）

硬链接是指将多个文件名关联到同一个文件的**i-node**，使得这些文件名共享同一个文件数据。

特点：

1. **硬链接不是独立的文件**，而是和目标文件共享相同的 i-node。也就是说，硬链接和原文件之间是**完全平等的**。
2. 硬链接**指向文件的 i-node**，而不是文件名或路径。因此，即使原文件被删除，硬链接仍然可以访问该文件的数据。
3. 文件的硬链接数反映了有多少个文件名指向同一个文件。当文件的所有硬链接都被删除时，文件的数据才会真正被删除。
4. 硬链接只能指向文件，**不能指向目录**。
5. 硬链接只能在**同一文件系统**内创建，不能跨文件系统。

硬链接的使用场景：

- 硬链接通常用于确保文件的多副本可以同步更新。当任意一个硬链接修改文件时，其他所有链接的内容也会同步变化。
- 可以用硬链接创建同一文件的多个路径引用，从而提供不同的访问方式。

硬链接的优缺点：

- **优点**：
  - 文件名删除后，硬链接仍然可以访问文件的数据。
  - 不占用额外的存储空间，因为它不需要创建新的文件，只是增加了文件的引用。
- **缺点**：
  - 不能跨文件系统创建硬链接。
  - 不能对目录创建硬链接（为了避免文件系统循环引用问题）。

open函数的应用

1

功能介绍

open函数主要用于打开一个指定的文件或设备，它会返回一个文件描述符，通过该描述符可以对文件执行后续操作。

2

函数原型

原型定义：

```
int open(const char *pathname, int flags, mode_t mode);
```

它接受文件路径、文件打开模式和文件权限模式作为参数。

3

参数解析

pathname是文件或设备的路径，flags定义了文件的打开模式（如只读或读写），mode则在创建新文件时定义文件的权限。

read函数详解

函数原型

`ssize_t read(int fd, void *buf, size_t count);`

用于指定从哪个文件读取，以及读取的字节数。

参数详解

fd是文件描述符，buf是指向数据缓冲区的指针，而count则表明要读取的最大字节数。

返回值是一个`ssize_t`类型的整数，它表示成功读取的字节数或者出现错误时的返回值。

错误：

如果读取过程中出现错误，返回-1，并设置全局变量`errno`来指示出现的具体错误类型。

write函数的工作原理

函数功能

write函数用于将数据写入由文件描述符指代的文件中，这是较为底层的数据写操作。

函数原型

ssize_t write(int fd, const void *buf, size_t count);

用于执行数据写入。

重要参数

在这里，fd依旧是文件描述符，buf是源数据的缓冲区地址，count描述了写入的字节数。

错误：

如果读取过程中出现错误，返回-1

利用stat函数获取文件状态

1

功能概述

stat函数能够提供文件的详细状态信息，包括文件大小、权限、修改时间等。

2

原型说明

```
int stat(const char *pathname, struct stat *statbuf);
```

专门用来获取文件属性。

3

参数详解

pathname参数接收文件路径，而statbuf参数则是一个指向stat结构的指针，用来存储获取到的文件状态。

目录操作函数深入

打开目录流

opendir函数用于打开一个目录流，它让我们可以读取目录中的文件列表。

读取目录内容

readdir函数可以从打开的目录流中读取一个目录项的信息，使我们能够遍历整个目录。

关闭目录流

closedir函数用于关闭一个已打开的目录流，这是一种良好的系统资源管理习惯。

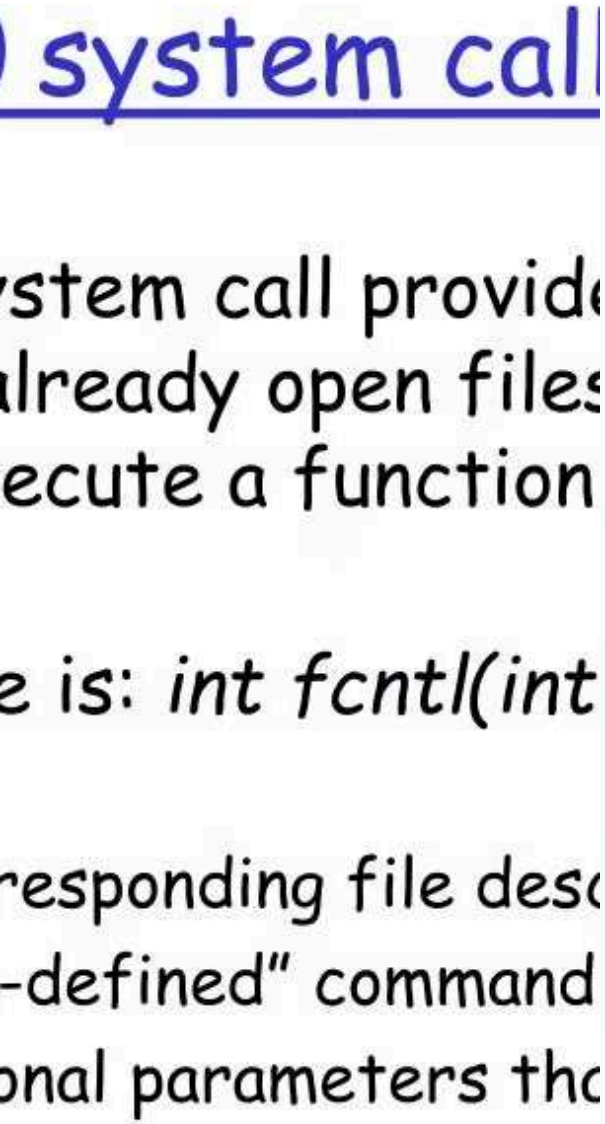
文件描述符复制——dup和dup2

dup函数

dup函数通过创建一个新的文件描述符，来复制一个给定的文件描述符。它常常用于文件描述符的重定向操作。

dup2函数

dup2函数不仅复制文件描述符，还允许用户指定新文件描述符的值。如果新文件描述符已经存在，dup2会先关闭它，再进行复制。



fcntl函数

1

改变文件性质

fcntl函数是一个用来改变已经打开的文件属性的强大工具。例如，你可以使用它来控制文件描述符的传承性。

2

函数原型

```
int fcntl(int fd, int cmd, .../* arg */);
```

通过传入不同的cmd参数，可以执行不同的操作。

3

锁定和其他操作

这些操作包括对文件加锁，获取或设置文件描述符的标志等，极大地提高了I/O操作的灵活性。

Linux 开发环境

Linux开发环境

1 **GCC编译器概述**

GCC作为Linux下的核心开发工具，支持多种编程语言，包括C和C++

2 **Makefile的作用**

Makefile定义了编译和链接程序所遵循的规则，是自动化构建流程中不可或缺的部分。

3 **GDB调试工具**

掌握GDB调试工具的使用可以帮助你更快地定位和解决代码中的问题。

4 **虚拟地址空间**

了解Linux如何为每个进程提供独立的虚拟地址空间，它对于内存管理和程序运行至关重要。

GCC编译

C程序编译

1

使用gcc命令来编译C源文件，包括命令的格式及其选项。

C++程序编译

2

介绍了g++命令专门用于编译C++源文件，并比较了它与gcc的不同之处。

编译选项

3

探讨了GCC提供的多种编译选项，如优化级别、调试信息生成、链接库等，以及它们的具体应用。

编译C程序

- gcc命令：用于编译C语言源文件。
- 示例： `gcc program.c -o program`
  - program.c：C语言源代码文件。
  - -o program：指定编译后的输出文件名为program。
- 解释：
  - gcc是GCC中用于编译C程序的命令。
  - -o选项用于指定输出的可执行文件的名称，如果不使用-o，默认生成的可执行文件名为a.out。

g++ 介绍

g++ 是 GCC（GNU Compiler Collection，GNU 编译器套件）中的一个用于编译 C++ 代码的编译器。它是开源的，并且支持多种平台，是开发 C++ 程序的常用工具。g++ 主要用于将 C++ 源代码（通常是 .cpp 文件）编译为目标文件（如 .o 文件）或可执行文件。

g++ 作为 GCC 的一部分，支持编译、链接、调试信息生成、预处理等操作。它不仅可以处理 C++ 语言标准，还支持各种编译优化和调试选项，是 C++ 开发的重要工具。

G++ 编译器的底层是用 **C 和 C++** 实现的，部分可能会使用汇编语言来实现底层硬件相关的优化

用 C++编译C++？

编译器的自举（Bootstrapping）

自举是编译器开发中一个重要的技术，指的是使用已有的编译器来编译新的编译器版本。也就是说，最早的编译器并不是用高级语言（如 C++）编写的，而是用汇编语言或其他低级语言编写的。随着编译器的演进，可以逐渐使用更高级的语言（如 C++）来重写编译器。

自举的基本步骤：

1. **初始编译器**：最早的编译器可能是用汇编语言或其他低级语言编写的。这些早期的编译器可以用来编译用 C 或 C++ 编写的编译器。
2. **用低级语言编写早期版本的编译器**：例如，最早的 C++ 编译器是用 C 语言编写的。这个编译器可以将 C++ 源代码编译成可执行文件。
3. **自举过程**：一旦有了第一个能编译 C++ 的编译器，就可以使用这个编译器来编译用 C++ 语言编写的编译器代码。这个过程称为**自举**，因为新编译器是通过编译旧编译器的方式生成的。
4. **重复过程**：随着编译器的不断优化和更新，每次新版本的编译器都可以通过旧版本编译出来。这种过程可以一直持续，编译器会变得越来越强大。

编译选项

- 优化级别（-O）：GCC提供了多种优化级别，例如-O0（无优化）、-O1（一般优化）、-O2（更多优化）、-O3（最高级别优化）。
- 优化级别越高，运行起来越快，但编译时间越长，编译后的代码越长
- 调试信息（-g）：加入-g选项，GCC会在编译的可执行文件中包含程序执行过程中的调试信息，便于使用调试工具（如gdb）进行调试。
- 链接库（-l和-L）：
  - -l：用于指定编译时需要链接的库，如-lm表示链接数学库libm。
  - -L：用于指定库文件的搜索路径。
  - gcc -o my_program my_program.c -lm -L/path/to/libm

Makefile构建示例

Makefile的定义

Makefile文件定义了一组规则，用于指定如何编译、链接和生成程序，它是自动化构建程序的核心文件。

Makefile定义了编译源代码和生成目标文件（如可执行文件或库）的步骤。它可以简化编译过程，特别是对于包含多个源文件的大型项目。

Makefile示例

```
目标: 依赖
    命令
all: program。

program: program.o
    gcc program.o -o program

program.o: program.c
    gcc -c program.c

clean:
    rm -f program program.o
```

规则

- **all规则**：
  - 目标all通常作为默认目标，它依赖于program。
  - 执行make命令时，默认执行all规则，进而编译生成program。
- **program规则**：
  - 目标program依赖于program.o。
  - 命令gcc program.o -o program用于链接对象文件program.o，生成可执行文件program。
- **program.o规则**：
  - 目标program.o依赖于源文件program.c。
  - 命令gcc -c program.c用于编译program.c，生成对象文件program.o。
- **clean规则**：
  - 目标clean没有依赖。
  - 命令rm -f program program.o用于清理编译生成的文件，-f选项表示强制删除。

Makefile语法

- **Tab缩进**：每个规则中的命令必须以Tab字符开始。
- **变量**：可以定义变量来简化Makefile，如`CC = gcc`，然后使用`$(CC)`来引用变量。
- **注释**：使用`#`开始注释行。
- **通配符**：支持使用通配符（如`*.c`）匹配文件名。
- **条件判断**：可以根据条件执行不同的命令

GDB调试基础

- 定义：GDB（GNU Debugger）是一个强大的Unix/Linux下的程序调试工具。
- 功能：可以用来跟踪程序执行过程，检查程序中的错误。

GDB（GNU Debugger）

GDB是一个强大的源代码级调试器，主要用于C、C++和其他支持的语言。它通过以下几点工作原理：

1 符号表和调试信息

编译器会生成包含调试信息的可执行文件，包括变量名、函数名、行号等。

2 进程控制

GDB与操作系统紧密合作，在目标进程上设置断点并暂停、恢复执行、单步跟踪等操作。

3 内存和寄存器访问

GDB可以读取并修改进程的内存和CPU寄存器内容，允许开发者检查或改变程序状态。

4 栈回溯

GDB分析堆栈帧以确定函数调用链，显示每个函数调用的局部变量值和返回地址。

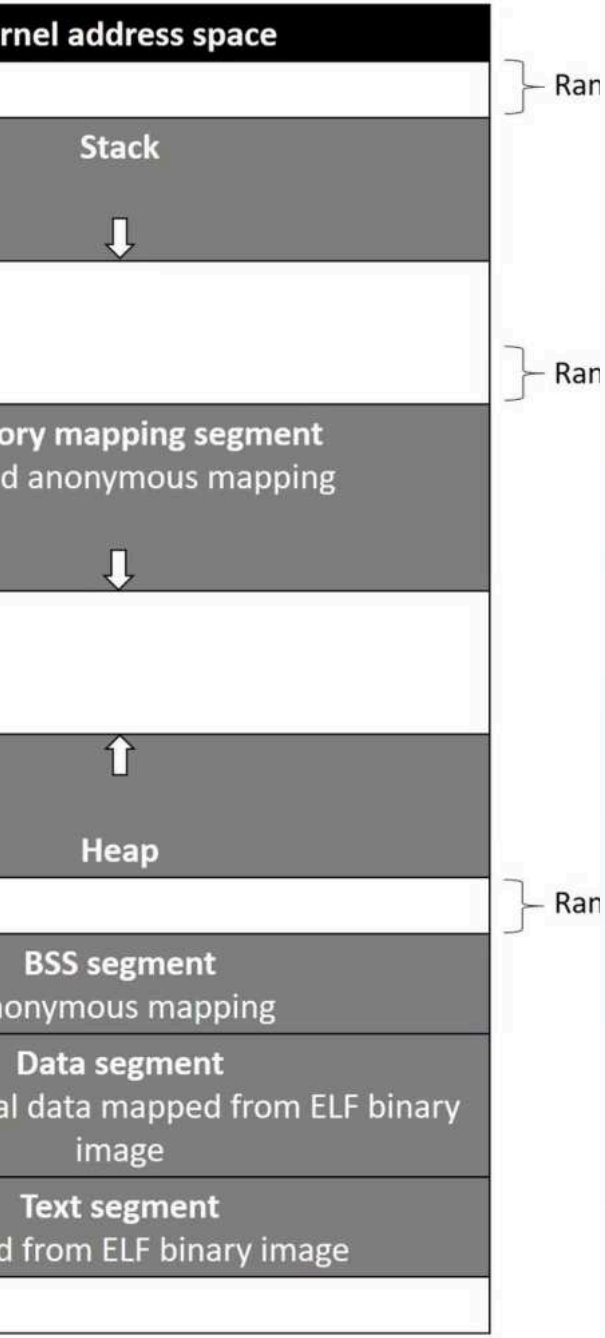
5 动态加载库的支持

GDB能够处理依赖动态链接库的程序，追踪到库中的函数并提供调试信息。

GDB调试基础

以下是GDB调试工具的基本使用方法：

1. 启动GDB：
2. 设置断点：
3. 运行程序：
4. 查看变量：
5. 单步执行：
    - next（n）：执行下一行代码，如果下一行是函数调用，则整个函数体将被执行。
    - step（s）：单步执行，如果遇到函数调用，将进入该函数内部。
6. 继续执行：
7. 查看堆栈信息：
8. 附加到正在运行的进程：



理解Linux虚拟地址空间

1 独立内存抽象

每个进程在Linux中具有独立的虚拟地址空间，允许程序在彼此隔离的环境中运行。

2 内存保护

虚拟地址空间机制提供了内存保护功能，防止进程间相互干扰，确保系统的稳定性。

3 地址隔离

通过给每个进程提供独立地址空间，Linux确保了数据的隔离性和安全性。

Linux 常见命令

Linux操作系统常用命令（面试不考，但得了解）

- ls：列出目录内容
- cd：改变目录
- cp：复制文件或目录
- mv：移动或重命名文件或目录
- rm：删除文件或目录
- grep：文本搜索工具
- find：查找文件
- cat：查看文件内容
- echo：显示一行文本
- chmod：改变文件权限
- man：查看命令手册

软件架构基础

软件架构是关于软件系统的高层结构和组件之间的关系，影响软件的性能、可维护性和可扩展性

C/S架构（Client/Server）

C/S架构 是一种常见的计算机网络架构，它将应用程序分为两个主要组成部分：客户端和服务器。

- **客户端（Client）**：客户端是用户或应用程序的界面，它通过网络连接到远程服务器，并向服务器发送请求。客户端通常负责用户交互和前端呈现。
- **服务器（Server）**：服务器是一个专用的计算机或应用程序，它接收来自客户端的请求，处理这些请求，并返回相应的数据或服务。服务器通常运行在后台，提供服务和资源。

特点

- **中心化管理：服务器充当中心，负责存储和管理数据，客户端通过请求来访问数据。**
- **高度可控性**：服务器可以实施访问控制和安全策略，确保数据的安全性和完整性。
- **适用于复杂任务**：C/S架构适用于需要大量计算或处理的复杂任务，因为服务器通常具有更强大的计算能力。
- **适用于跨平台**：客户端和服务器可以运行在不同的操作系统和硬件平台上。

P2P架构（Peer-to-Peer）

P2P架构 是一种分布式计算架构，其中每个节点（通常是计算机或设备）都具有相同的地位，即既是客户端又是服务器。节点之间可以直接通信，而不需要中心化的服务器。

特点

- **去中心化**：P2P网络没有中心服务器，每个节点都可以直接与其他节点通信，共享资源和服务。
- **资源共享**：P2P网络允许节点共享文件、带宽、计算能力和其他资源，通常用于文件分享、流媒体传输等。
- **分布式控制**：每个节点可以自行决定如何分配资源和处理请求，而不依赖单一的中心控制。
- **鲁棒性**：P2P网络通常具有很强的鲁棒性，因为没有单一故障点，网络可以继续工作即使部分节点不可用。
- **广泛应用**：P2P架构常用于文件共享应用程序（如BitTorrent）、即时通信（如Skype）和区块链等领域。

BS和CS架构模式

- **BS架构（浏览器-服务器架构）**
  - 特点：客户端通常是Web浏览器，通过HTTP协议与服务器通信。
  - 应用：网页浏览、在线应用等。

B/S架构是一种特定类型的C/S架构，其中客户端是一个Web浏览器，它向Web服务器发送HTTP请求并接收HTML、CSS、JavaScript等资源作为响应。用户界面在浏览器端渲染和执行，而大部分业务逻辑和数据处理在服务器端完成。相比传统的桌面应用C/S架构，B/S架构具有跨平台性好、客户端零安装、更新方便等特点

- **CS架构（客户端-服务器架构）**
  - 特点：客户端和服务器是两个独立的应用程序，通过网络直接通信。
  - 应用：电子邮件、网络游戏、文件传输等。

Linux面试常见问题

问题1：Linux和Unix的区别是什么？

- **Unix**：

- Unix 本身是一组操作系统标准和规范，可以由不同的公司或组织实现。这意味着**不同的 Unix 版本**（如 AIX、HP-UX、Solaris、BSD）可能具有不同的实现细节和特性。Unix 通常是一个**商用产品**，且大部分 Unix 操作系统都是基于**封闭源代码**

- **Linux**：

- Linux 是严格意义上的操作系统内核。内核管理硬件资源、提供系统调用等，用户通常会通过**Linux 发行版**（如 Ubuntu、Red Hat、Debian）来使用完整的 Linux 系统。与 Unix 不同，Linux 是一个**完全开源**的项目，遵循**GPL**（GNU General Public License）许可证

问题2：解释一下Linux的文件权限（rwx）的含义。

回答：

- **r（读，Read）**：允许查看文件内容或列出目录内容。

- **w（写，Write）**：允许修改文件内容或在目录中添加/删除文件。

- **x（执行，Execute）**：允许执行文件或访问目录。

权限分为三类：

- **用户（Owner）**：文件的所有者。

- **组（Group）**：与文件所有者同组的用户。

- **其他（Others）**：所有其他用户。

问题3：解释Linux中的软链接和硬链接的区别。

回答：

- **硬链接**：

- 直接指向文件的inode。
- 不能跨文件系统。
- 删除原文件不会影响硬链接，文件实际删除仅在所有链接被删除后。

- **软链接（符号链接）**：

- 作为指向原文件路径的快捷方式。
- 可以跨文件系统。
- 删除原文件后，软链接将变为断链。

属性	软链接（Symbolic Link）	硬链接（Hard Link）
i-node 关系	软链接有自己独立的 i-node，指向目标文件	硬链接和目标文件共享同一个 i-node
指向	指向文件路径（类似快捷方式）	指向文件本身（通过 i-node）
文件或目录	可以指向文件或目录	只能指向文件，不能指向目录
跨文件系统	可以跨文件系统	不能跨文件系统
目标文件删除后的行为	软链接失效，成为“悬空链接”	文件仍然可访问，直到所有硬链接都被删除
空间使用	占用少量空间，用于存储路径信息	不占用额外空间，只是增加了 i-node 引用

谢谢大家