第4课 Linux基础知识

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。

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(这部分不用记，但得了解了解)

历史背景：

Linux由Linus Torvalds于1991年创建，作为Unix操作系统的一个开源替代品。

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。

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。

跨平台： Linux能在各种不同架构的硬件平台上运行，从嵌入式设备到大型服务器集群，甚至是超级计算机

Linux特性

1. **一切皆文件（Everything is a file）： 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。这意味着你可以对它们进行读写操作，就像对待普通文件一样。例如，硬件设备可以通过特殊的设备文件来访问和控制。**
2. 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。
3. 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。
4. 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

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

1. **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）。
2. **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。
3. **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。
4. **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

1. **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。
2. **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。
3. **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。
4. **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文件操作与管理

1. 文件描述符

- **定义**：文件描述符是一个抽象指标（一个非负整数），用于表示对文件或其他I/O资源的访问。
- 详细定义：在Unix-like操作系统（如Linux）中，文件描述符（File Descriptor, FD）是一个非负整数，它是系统内核用于标识一个已打开的文件、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，**通过这张表，进程可以访问和管理其拥有的所有打开的I/O资源
- 不同资源有不同的文件描述符：在同一进程中，不同的文件或者资源（如设备文件、网络套接字）会分配不同的文件描述符，**确保每个描述符对应着一个独立的I/O资源**
- 文件描述符表：Linux内核为每个进程维护了一个文件描述符表，这个表记录了每个描述符关联的具体资源以及当前的状态信息（如读写位置、权限等）。当进程通过系统调用操作一个文件描述符时，内核会根据描述符查找到对应的资源并执行相应的操作。
    - 简单理解就是下图这样一个数组，文件描述符（索引）就是文件描述符表这个数组的下标，数组的内容就是指向一个个打开的文件的指针。

上面只是简单理解，实际上关于文件描述符，Linux内核维护了3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

一个 Linux 进程启动后，会在内核空间中创建一个 PCB 控制块，PCB 内部有一个文件描述符表（File descriptor table），记录着当前进程所有可用的文件描述符，也即当前进程所有打开的文件。进程级的描述符表的每一条记录了单个进程所使用的文件描述符的相关信息，进程之间相互独立，一个进程使用了文件描述符**3**，另一个进程也可以用**3**。

除了进程级的文件描述符表，系统还需要维护另外两张表：打开文件表、i-node 表。这两张表存储了每个打开文件的打开文件句柄（open file handle）。一个打开文件句柄存储了与一个打开文件相关的全部信息。

- 打开文件表（Open File Table）：记录了进程如何使用特定的打开文件，如当前读写位置、访问模式等。这样可以快速定位到进程对文件的具体操作状态，提高系统处理I/O请求的效率。
- i-node 表：存储的是文件本身的元数据信息，包括但不限于大小、权限、时间戳以及指向实际数据块的指针。这些信息对于管理文件系统中的所有文件是通用且持久的，不依赖于任何特定进程。

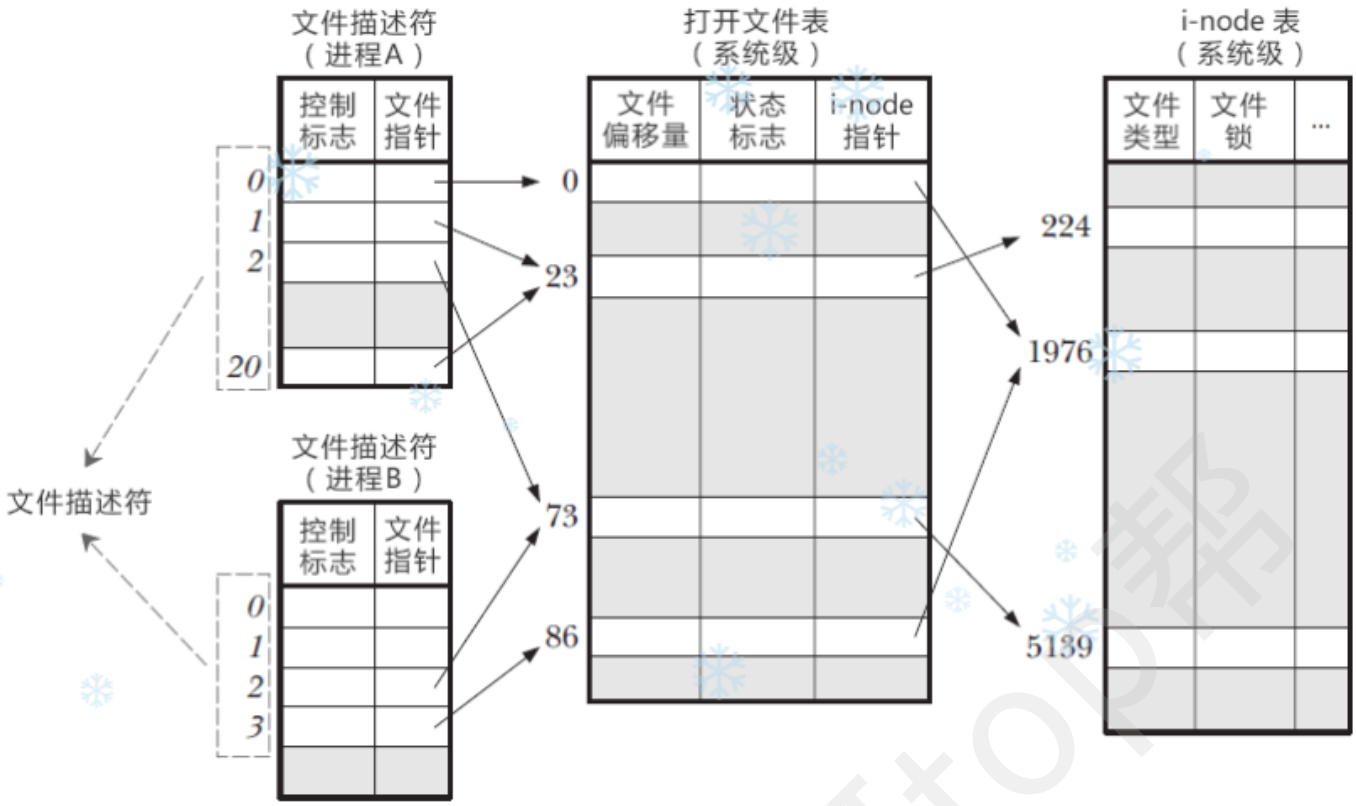
(下面的原理仅用于了解，无需记忆)

系统级的打开文件描述符表：

- 当前文件偏移量（调用read()和write()时更新，或使用lseek()直接修改）
- 打开文件时的标识（open()的flags参数）
- 文件访问模式（如调用open()时所设置的只读模式、只写模式或读写模式）
- 与信号驱动相关的设置
- 对该文件i-node对象的引用，即i-node 表指针

文件系统的i-node表：

- 文件类型（例如：常规文件、套接字或FIFO）和访问权限
- 一个指针，指向该文件所持有的锁列表
- 文件的各种属性，包括文件大小以及与不同类型操作相关的时间戳



- 在进程 A 中，文件描述符 1 和 20 都指向了同一个打开文件表项，标号为 23（指向了打开文件表中下标为 23 的数组元素），这可能是通过调用 dup()、dup2()、fcntl() 或者对同一个文件多次调用了 open() 函数形成的。
- 进程 A 的文件描述符 2 和进程 B 的文件描述符 2 都指向了同一个文件，这可能是在调用 fork() 后出现的（即进程 A、B 是父子进程关系），或者是不同的进程独自去调用 open() 函数打开了同一个文件，此时进程内部的描述符正好分配到与其他进程打开该文件的描述符一样。
- 进程 A 的描述符 0 和进程 B 的描述符 3 分别指向不同的打开文件表项，但这些表项均指向 i-node 表的同一个条目（标号为 1976）；换言之，它们指向了同一个文件。发生这种情况是因为每个进程各自对同一个文件发起了 open() 调用。同一个进程两次打开同一个文件，也会发生类似情况。

文件描述符的特性

- 唯一性：在一个进程中，每个打开的资源都有一个唯一的文件描述符与之关联。
- 标准文件描述符：
  - 标准输入 (stdin) 对应文件描述符0
  - 标准输出 (stdout) 对应文件描述符1
  - 标准错误输出 (stderr) 对应文件描述符2
- 继承性：子进程通常会继承父进程的一部分或全部文件描述符。

2. open函数：如何获取文件描述符（这些函数看看理解就行，都不需要记住）

- **功能**：用于打开文件或设备，返回一个文件描述符。

- **原型**：

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

- **参数**：
  - `pathname`：要打开的文件路径。
  - `flags`：打开文件的模式，如只读（`O_RDONLY`）、只写（`O_WRONLY`）、读写（`O_RDWR`）等。
  - `mode`：设置新文件的权限，仅在创建新文件时使用。

使用文件描述符操作文件

3. read函数

- **功能**：从文件描述符指向的文件中读取数据。
- **原型**：

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

- **参数**：
  - `fd`：文件描述符。
  - `buf`：数据读取后存放的缓冲区地址。
  - `count`：要读取的字节数。
  - 返回值：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。如：
  - `EINTR`：读取操作被信号中断。
  - `EFAULT`：`buf`指针无效，即指向的内存不可访问。
  - `EIO`：发生硬件I/O错误。
  - `EISDIR`：试图从目录文件中读取数据

4. write函数

- **功能**：向文件描述符指向的文件写入数据。
- **原型**：

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

- **参数**：
  - `fd`：文件描述符。
  - `buf`：要写入文件的数据的缓冲区地址。
  - `count`：要写入的字节数。

5. lseek函数

- **功能**：重新定位文件描述符的文件偏移量。
- **原型**：

```
1 off_t lseek(int fd, off_t offset, int whence);
```

- **参数**：
  - `fd`：文件描述符。
  - `offset`：相对偏移量。
  - `whence`：偏移量的起始位置，如文件开头（`SEEK_SET`）、当前位置（`SEEK_CUR`）、文件末尾（`SEEK_END`）。

6. stat函数

- **功能**：获取文件的状态信息。
- **原型**：

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

- **参数**：
  - `pathname`：文件路径。
  - `statbuf`：`stat`结构体，存储获取到的文件信息。

7. 目录操作函数

- **功能**：提供了一系列操作目录的函数，如：
  - `opendir`：打开一个目录流。
  - `readdir`：读取目录流中的下一个目录项。
  - `closedir`：关闭目录流。

8. dup函数和dup2函数

- **dup函数**：用于复制文件描述符。
  - `dup`：创建一个新的文件描述符，复制指定的文件描述符。
- **dup2函数**：与 `dup` 类似，但可以指定新的文件描述符值。

9. fcntl函数

- **功能**：改变已打开的文件的性质。
- **原型**：

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

- **用途**：包括改变文件描述符的标志、对文件加锁等。

通过学习这些基础的文件操作和管理函数，可以在Linux环境下更有效地进行文件和目录的处理，从而支持复杂的应用程序开发和数据管理。

Linux开发环境搭建教学内容

1. GCC编译

GCC简介

- GCC定义：GCC，全称GNU Compiler Collection（GNU编译器套件），是一套功能强大的编程语言编译器。
- 功能：GCC支持多种编程语言，包括C、C++、Objective-C、Fortran、Ada等。它能够编译源代码、生成目标代码，最终产生可执行文件。

- 应用：GCC是Linux系统下最常用的编译器之一，广泛应用于软件开发和系统编程。

编译C程序

- gcc命令：用于编译C语言源文件。
- 示例：

```
1 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`。

编译C++程序

- g++命令：用于编译C++语言源文件。
- 示例：

```
1 g++ program.cpp -o program
```

  - `program.cpp`：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：
  - `g++`是GCC中专门用于编译C++程序的命令。
  - 与`gcc`相似，`g++`也支持`-o`选项来指定输出的可执行文件名。

编译选项

只需记住：**优化级别越高，编译过程越慢，编译出来的程序越快**

- 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）。

  以下是常见的 GCC 优化级别和其含义：

  a. -O0（无优化）：

- 此级别下，编译器不进行任何优化，生成的代码与源代码基本相同，主要用于调试和分析目的。
- 编译速度较快，但生成的代码可能较慢且占用更多的内存空间。

b. -O1（一般优化）：

- 这个级别进行一些基本的优化，如删除未使用的变量、函数内联、循环展开等。
- 生成的代码会比无优化级别更高效，但编译速度仍然较快。

c. -O2（更多优化）：

- 在这个级别下，编译器会执行更多的优化，包括更强大的循环优化、内联函数的更多展开、更多的寄存器分配等。
- 这会导致生成的代码更快，但编译时间可能会增加。

d. -O3（最高级别优化）：

- 这个级别启用了几乎所有可能的优化，包括更复杂的循环优化、矢量化优化等。
- 生成的代码在性能上通常最好，但编译时间可能显著增加。

- 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）：链接库是包含已编译代码的文件，可以在程序中重复使用。它们通常包含一组函数或功能，供程序调用。当程序需要使用链接库中的功能时，链接器会在可执行文件中创建引用链接，而不是复制整个功能的代码，这样可以大大减小可执行文件的大小，因为不需要将这些功能的代码重复包含在每个可执行文件中
  - `-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。

```
1 gcc -o my_program my_program.c -lm -L/path/to/libm
```

2. 静态库和动态库

- **静态库**：静态库（`.a` 文件）**在程序编译时被完整地复制到可执行文件中**。在链接阶段，当程序与静态库进行链接时，链接器会将静态库中的所有代码和数据完整地复制并整合到最终生成的可执行文件中。
  - 优点：

        - 独立性：由于静态库的所有内容都被嵌入到了可执行文件中，因此该程序可以在没有外部依赖的情况下独立运行。
        - 执行速度：因为函数调用直接在可执行文件内部完成，理论上减少了动态加载库的时间开销。
    - 缺点：
        - 可执行文件体积较大：包含静态库代码的可执行文件通常比仅引用动态库的可执行文件大得多，特别是在库很大或者多个程序都使用同一个库时，这种空间浪费尤为明显。
        - **更新困难：如果静态库有更新，需要重新编译所有依赖它的程序才能应用新的改动。**

- **动态库**：动态库（`.so` 文件）在程序运行时被加载。**动态库不会被复制到可执行文件中，可执行文件只包含对动态库的引用。动态库可以被多个程序共享，节省了磁盘空间**，但部署时需要确保动态库在系统上可用。
    - 优点：
        - 节省内存和磁盘空间：多个程序可以共享同一份动态库文件，从而节省存储资源。
        - 动态加载和更新：程序在启动时或运行过程中按需加载库，且只需更新库文件即可影响所有依赖它的程序，无需重新编译每个程序。
    - 缺点：
        - 部署复杂：部署一个依赖动态库的程序时，必须确保相应的动态库存在于系统的正确路径上，并且版本兼容。
        - 性能影响：动态库的加载可能会带来额外的时间开销，尤其是在首次加载或库发生变化时。
        - 兼容性和稳定性问题：如果系统环境缺少特定版本的动态库，或者动态库存在bug，可能会影响程序的正常运行。

3. Makefile

（这个也是不用记，面试不会问，但属于C++开发需要掌握的基本技能）

Makefile 是一种定义了一组规则来指定如何编译、链接和生成程序的文件。一个基本的Makefile示例如下：

```
all: program

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

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

```
8
9  clean:
10     rm -f program program.o
```

- **伪目标**：`all` 和 `clean` 是伪目标，它们不代表文件，而是规则的名字。
- **依赖关系**：`program` 依赖于 `program.o`，`program.o` 依赖于 `program.c`。
- **规则**：每个规则后的行定义了如何生成目标文件，例如用 `gcc` 来编译 `.c` 文件或链接 `.o` 文件。
- **清理**：`clean` 规则用于删除编译过程中产生的文件。
- 规则：Makefile由一系列的规则构成。每个规则的格式通常为：

```
1  目标：依赖
2      命令
```

- 其中“目标”是要生成的文件，“依赖”是生成目标所需的文件或条件，“命令”是生成目标的具体命令。
- 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语法规则

- 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

4. GDB调试

GDB 是Linux下强大的程序调试工具。基本的GDB使用流程包括设置断点、单步执行代码、查看变量等。

```
1 gdb program
2 (gdb) break main
3 (gdb) run
4 (gdb) print variable
5 (gdb) continue
```

GDB调试工具的原理：

GDB（GNU Debugger）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。GDB的工作原理主要基于以下几点：

1. **符号表和调试信息**：

当使用`-g`选项编译程序时，编译器会生成包含调试信息的可执行文件，这些信息包括变量名、函数名、行号等。GDB通过读取这些调试信息来理解程序结构。

2. 进程控制：

GDB与操作系统紧密合作，能够对被调试的进程进行暂停、恢复执行、单步跟踪等操作。它利用了操作系统的ptrace系统调用或其他类似机制，在目标进程上设置断点，并在指定位置暂停进程执行。

3. 内存和寄存器访问：

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

4. 栈回溯：

当程序因为异常停止时，GDB能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，这对于理解函数调用顺序和错误发生的位置至关重要。

5. 动态加载库的支持：

对于依赖动态链接库的程序，GDB能够处理动态加载的情况，追踪到库中的函数并提供相应的调试信息。

GDB调试工具的基本使用方法：

1. **启动GDB**：

通常使用命令 `gdb executable` 来启动GDB，其中executable是你的程序文件。

2. **设置断点**：

在源代码中某一行设置断点，可以使用命令 `break filename:linenumber` 或者 `break function_name`。

示例：

```
(gdb) break main.cpp:10
```

3. 运行程序：

使用 `run [args]` 命令运行程序，可以传递命令行参数给程序。

示例：

```
(gdb) run arg1 arg2
```

4. 查看变量：

在程序暂停时，可以通过 `print variable` 查看变量的当前值。

示例：

```
1 (gdb) print myVariable
```

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：

使用 `continue`（c）命令继续执行程序，直到遇到下一个断点或者程序结束。

7. 查看堆栈信息：

使用 `backtrace`（bt）命令查看调用堆栈，了解函数调用层级及各层的局部变量情况。

8. 附加到正在运行的进程：

如果需要调试一个已经运行的进程，可以使用 `attach process_id` 命令将其附加到GDB进行调试。

以上只是GDB功能的一部分，实际应用中还有许多其他高级特性，例如监视表达式、设置条件断点、查看内存布局等。

5. 虚拟地址空间

每个进程在Linux中拥有独立的虚拟地址空间，是对物理内存的抽象。这提供了内存保护和地址隔离的功能。

（408学的虚拟内存知识）

虚拟地址空间的主要特点和优势包括：

1. 隔离性（Isolation）：
    - 每个进程都拥有自己独立的虚拟地址空间，这意味着**一个进程无法直接访问其他进程的内存区域**，从而避免了因一个进程错误操作而导致其他进程数据受损的问题，提高了系统的稳定性与安全性。
2. 内存保护（Protection）：

- **虚拟地址空间通过页表机制（如Linux中的MMU和页表）将虚拟地址映射到物理地址。**系统可以设置这些映射关系的不同权限（读、写、执行），防止非法访问或修改，比如只允许程序访问其自己的栈、堆以及已分配的内存区域，而不能随意访问内核或其他进程的数据。

3. 资源有效利用（Efficient Resource Utilization）：
   - 通过虚拟内存技术，即使物理内存不足，也可以通过交换空间（swap space）暂时将部分进程数据换出到磁盘上，从而让更多的进程能够同时运行，提升了内存资源的有效利用率。
4. 内存共享（Memory Sharing）：
   - 尽管每个进程有独立的虚拟地址空间，但多个进程可以选择共享某些内存区域，例如，动态链接库（.so文件）被所有使用它的进程以只读方式共享，节省了物理内存资源。
5. 地址重映射（Address Remapping）：
   - 系统可以在不同的时间点根据需要动态地改变虚拟地址到物理地址的映射，这使得进程能够在不需要更改代码的情况下就能获得更大的可用内存空间，或者进行内存优化操作，例如内存分页和页面替换等。
6. 简化编程（Simpler Programming）：
   - 对于程序员来说，只需要关注虚拟地址而非物理地址，可以编写更加简洁且平台无关的代码，不必关心具体的物理内存布局及碎片问题。

Linux操作系统常用命令（面试不考，但得会用）

1. **文件和目录操作**：
   - cd：改变当前目录。
   - mkdir：创建新目录。
   - rmdir：删除空目录。
2. **系统管理命令**：
   - top：实时监控系统资源使用情况。
   - ps：查看当前运行的进程。
   - kill：终止进程。
3. **网络命令**：
   - ping：检查网络连接。
   - ifconfig：查看网络接口配置。
4. **用户管理命令**：
   - useradd：添加新用户。

- passwd：修改用户密码。

Git版本控制基础（不考，开发迟早得掌握）

Git是一个分布式版本控制系统，用于跟踪文件的变更和协调多人之间的工作。常用命令包括：

- `git clone`：克隆仓库
- `git add`：添加文件到暂存区
- `git commit`：提交更改
- `git push`：推送到远程仓库
- `git pull`：从远程仓库拉取更新
- `git branch`：管理分支
- `git merge`：合并分支

下面要记一下了

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。

我们的项目就是**BS架构**

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。

常见的面试中关于Linux系统的问题和答案

问题1：Linux和Unix的区别是什么？

回答：

- **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。

总结

特性	Unix	Linux
起源	1969 年 AT&T 贝尔实验室	1991 年 林纳斯·托瓦兹
源代码	大多数是专有、闭源的；一些是开源的（如 BSD）	完全开源，遵循 GPL 许可证
体系	商业产品，标准不同，版本多样	开源内核，组成完整系统的 Linux 发行版众多
应用场景	大型服务器、超级计算机、嵌入式系统	服务器、桌面、嵌入式、移动设备、云计算等
兼容性	遵循 POSIX 标准	遵循 POSIX 标准，与 Unix 高度兼容
稳定性	高，更新缓慢，维护成本高	高，可定制性强，社区支持，更新频繁

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。

问题4：如何查找系统中正在运行的进程？

回答：

- 使用 `ps` 命令：

```
1 ps aux
```

- 使用 `top` 命令：

```
1 top
```

- 使用 `htop` 命令（需安装）：

```
1 htop
```

- 使用 `pgrep` 命令：

```
1 pgrep process_name
```

问题5：如何终止一个正在运行的进程？

回答：

- 使用 `kill` 命令：

```
kill PID
```

- 发送默认的 `SIGTERM` 信号，优雅地终止进程。

- 使用 `kill -9` 命令：

```
kill -9 PID
```

- 发送 `SIGKILL` 信号，强制终止进程。

- 使用 `pkill` 命令（根据进程名）：

```
pkill process_name
```

问题6：如何在Linux中查找文件？

回答：

- 使用 `find` 命令：

```
find /path/to/search -name "filename"
```

- 按名称查找：

```
find /home -name "test.txt"
```

- 按类型查找（如目录、文件）：

```
find /home -type d -name "Documents"
```

- 按修改时间查找：

```
find /var/log -mtime -7
```

- **使用 `locate` 命令**（需先更新数据库）：

```
updatedb
locate filename
```

- **使用 `which` 命令**（查找可执行文件路径）：

```
which bash
```

问题7：解释Linux中的 `chmod` 。

回答：

- `chmod` （Change Mode）：

- 修改文件或目录的权限。
- 示例：给予所有者读、写、执行权限，组和其他用户读、执行权限。

```
chmod 755 filename
```

问题8：解释Linux中的 `chmod` 权限数值表示法。

回答：

- **权限数值**由三位数字组成，每位数字代表用户、组和其他用户的权限。
  - **r（读）=4**
  - **w（写）=2**
  - **x（执行）=1**

- **计算方法**：

- 将每类权限的数字相加，形成一个三位数。

- **示例**：

- `chmod 755 filename`
  - 用户权限：7 = 4+2+1 = rwx
  - 组权限：5 = 4+0+1 = r-x
  - 其他权限：5 = 4+0+1 = r-x

问题9：如何在Linux中查找正在监听的端口和对应的进程？

回答：

- **使用 `netstat` 命令**：

```
1 sudo netstat -tulnp
```

- `-t`：TCP连接
- `-u`：UDP连接
- `-l`：监听状态
- `-n`：显示数字形式的地址和端口
- `-p`：显示进程ID和程序名

- **使用 `ss` 命令**（更现代的替代品）：

```
sudo ss -tulnp
```

- **使用 `lsof` 命令**：

```
sudo lsof -i -P -n | grep LISTEN
```