上 海 大 学
2022-2023 秋季学期
《操作系统(1)（08305011）》课程考核报告

学 号: 20121034

姓 名: 胡才郁

课程考核评分表

序号	题号	分值	成绩
1	题目 1	20	
2	题目 2	20	
3	题目 3	20	
4	题目 4	20	
5	题目 5	20	
考核成绩		100	
评阅人签名			

计算机工程与科学学院

2022 年 11 月

题目 1：微内核与宏内核

1.1 微内核的定义

内核是管理系统资源的操作系统的核心部分。它是计算机应用程序和硬件之间的桥梁，内核在计算机 root 引导启动程序之后就会被加载。

而微内核是内核的分类之一，作为内核，它管理所有系统资源。不同于宏内核，在微内核中，**用户服务**和**内核服务**是在不同的地址空间中实现的。用户服务保存在**用户地址空间**，内核服务保存在**内核地址空间**，这样也减少了内核的大小和操作系统的大小。微内核设置了**用户态**与**内核态**两种状态，关于这两种状态之间的讨论将在题目 2 中详细展开。

微内核将核心功能**模组化**，划分成几个独立的模块，各自运行，形成服务。需要特权的进程，只有基本的线程管理，内存管理和进程间通信等等。这个部分，由一个简单的硬体抽象层与关键的系统调用组成，而其余的服务行程，则移至用户空间。

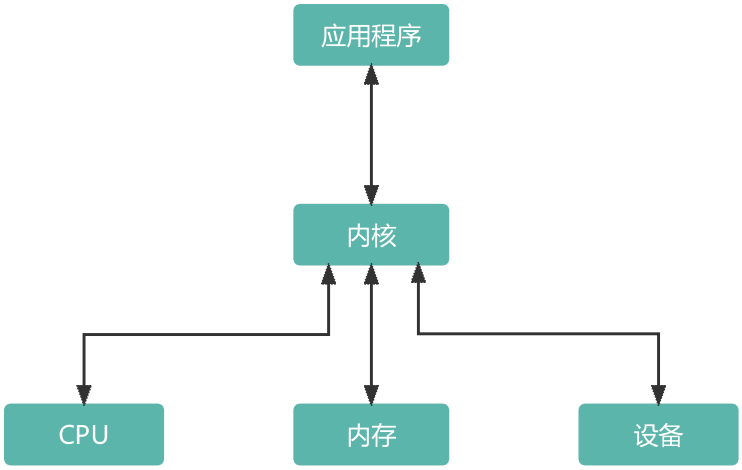

图 1 微内核架构

1.2 微内核的基本功能

由于内核是操作系统的核心部分，因此内核仅被用于处理最重要的服务。在这个微内核体系结构中，只有最重要的服务在内核中，其余的操作系统服务存在于系统应用程序中。微内核具体提供以下具体功能：

1. 进程管理

对于进程的调度而言，内核设置多个进程的优先级队列，将指定优先级的进程从进程队列中取出并运行。

对于进程的通信而言，由于所有服务进程都各自在不同地址空间内运行，因此在微核心架构下，不能像宏内核一样直接进行函数调用。在微核心架构下，微内核间建立进程间通讯机制，通过消息传递的方式来让服务进程间相互交换讯息，调用彼此的服务，以及完成同步等等。

2. 内存管理

微内核中配置最基本的低级存储器管理机制，例如实现将用户空间的逻辑地址变化为内存空间的物理地址的页表机制，以及地址变换机制等等，这些依赖于硬件的部分都存放在微内核中。

3. 中断处理

当中断发生时，内核会将中断捕获，进行对应的前期处理，如中断现场保护，识别中断类型等等。内核将中断信息转换为消息后发送给服务器，再由服务器调用相关程序进行处理。

1.3 微内核优点

1. 系统可扩展性强

微内核架构易于扩展，即如果要添加任何新服务，它们将被添加到用户地址空间，因此不需要在内核空间中进行修改。系统也可以根据需要，替换或新增某些服务进程，这种可“插拔”的结构更加灵活。

2. 系统的可靠性

微内核架构的各个服务间各自独立，减少系统之间的耦合度，易于调试与排错，也可增加可移植性。故障隔离和恢复可以避免单一服务失效而造成整个系统崩溃，即使某项服务出错，内核只需要重新启动这个服务即可，不致于影响到其他服务的正常运行，这样使得系统稳定度增加。尽管在用户地址空间中运行的 C/S 架构和服务之间的通信是通过消息传递建立的，降低了微内核的执行速度，但操作系统不受影响。因为用户服务和内核服务是隔离的，即使有用户服务失败，它不会影响内核服务。

3. 高复用性便于开发

微内核架构基于面向对象的思想，可重用了各个部件的功能。由于继承的存在，即使改变需求，维护也只是在局部模块。而其由于面向对象的继承、封装、多态的特性，自然设计出高内聚、低耦合的系统结构，使得系统更灵活、更容易扩展，并且成本较低。

1.4 微内核存在的问题

微内核最大的问题就在于**执行效率不高**，而效率不高的主要原因在于频繁进行**上下文切换**造成的性能损耗。

当由操作系统决定进行进程间的切换时，操作系统就会执行一些底层代码，即所谓的上下文切换。

上下文切换在概念上很简单：操作系统要做的就是为当前正在执行的进程保存一些寄存器的值，并为即将执行的进程恢复一些寄存器的值。这样一来，操作系统就可以确保最后执行 return-from-trap 指令时，不是返回到之前运行的进程，而是继续执行另一个进程。关于 return-from-trap 指令的详细作用，也会在问题 2 时详细展开。

为了保存当前正在运行的进程的上下文，操作系统会执行一些底层汇编代码，来保存通用寄存器、程序计数器，以及当前正在运行的进程的内核栈指针，然后恢复寄存器、程序计数器，并切换内核栈，供即将运行的进程使用。通过切换栈，内核在进入切换代码调用时，切换到被中断的进程的上下文；而在返回时，切换到即将执行的进程的上下文。当操作系统最终执行 return-from-trap 指令时，即将执行的进程变成了当前运行的进程。至此上下文切换完成。

时间片轮转的方式，使得多个作业利用一个 CPU 执行成为可能，但是保存现场和加载现场，也带来了性能消耗。完成一次客户对操作系统提出的服务请求时，需要利用消息实现多次用户态与核心态直接的切换。而如果上下文切换的次数过多，会导致 CPU 在操作系统控制下，并不能专注于计算，而像个“搬运工”，CPU

频繁在寄存器和进程队列之间转换。更多的时间花在了进程切换，而不是真正工作的进程上。

1.3 微内核与宏内核实例

1.3.1 Minix

Minix 是一种基于**微内核架构**的类 UNIX 计算机操作系统，是免费的开源操作系统。它很好的体现出了微内核的设计思想，即：高度可靠、灵活和安全。它基于在内核模式下运行的微型微内核，操作系统的其余部分作为许多隔离的、受保护的进程在用户态下运行。它可以运行在 x86 和 ARM CPU 上，并且与 NetBSD 兼容，运行着数以千计的 NetBSD 包。

全套 Minix 除了启动的部分以汇编语言编写以外，其他大部分都是用 C 语言编写。分为：核心、存储器管理及文件系统三部分。Minix 在设计之初，为了使程序简化，它将程序模块化，如文件系统与存储器管理，都不是在操作系统核心中运作，而是在用户空间运作。在 Minix 3 这个大版本时，连 IO 设备都被移到用户空间运作。

1.3.2 PikeOS

PikeOS 是一种商业硬实时操作系统，他也是**微内核架构**的典型代表。它提供基于分离内核的管理程序，具有多种逻辑分区类型。PikeOS 的一个关键特性是能够在同一计算平台上同时安全地执行具有不同安全级别的应用程序，这是通过软件分区对这些应用程序进行严格的空间和时间隔离来完成的。软件分区可以被视为具有预分配权限的容器，可以访问内存、CPU 时间、I/O 和预定义的操作系统服务列表。

PikeOS 是为航空航天、国防、汽车、运输、工业自动化、医疗、网络基础设施和消费电子领域具有认证需求的安全关键应用而开发的，使用户能够根据不同行业的高质量、安全和安保标准，为 IOT 物联网构建可认证的智能设备。

1.3.3 OpenBSD

OpenBSD 是一个以安全为中心的，免费开源的类 Unix 操作系统，基于 BSD。OpenBSD 使用**宏内核架构**，其开发团队为 OpenBSD 开发的软件包如 PF 防火墙、OpenSSH，在各种操作系统中都随处可见。OpenBSD 的安全增强功能、内置的加密功能和 PF 包过滤器使它在安全领域应用广泛，例如作为防火墙、入侵检测系统和虚拟专用网网关，OpenBSD 也因此被称为世界上最安全的操作系统。

图 2 OpenSSH

1.3.4 AIX

AIX 也是**宏内核**操作系统，它的名字来源于“先进交互执行系统”(Advanced Interactive executive)，AIX 级别的逻辑卷管理正逐渐被添加进各种自由的 UNIX 风格的操作系统中。值得一提的是，将 AI 应用于国际象棋，并曾经打败当时世界冠军的计算机深蓝使用的操作系统就是 AIX。

1.4 两种内核间的个人选择

就我个人而言，我会选择**微内核**。

关于体系结构中微内核和宏内核优劣的争论曾经非常激烈，两者间如何选择，也曾引发 Minix 与 Linux 作者之间的辩论，即著名的 Tanenbaum-Torvalds debate。在今天的许多通用操作系统中，Linux 仍采用宏内核，而 MacOSX、Windows 都采用混合内核设计。

在微内核设计中，所有的功能都被尽可能从内核中移除，而是交付给独立的操作空间，这些操作空间则通过进入内核态相互传递消息。而在宏内核设计中，内核和操作进程共享空间，消息在进程之间直接传递，而不需要在内核之间进行。形象的说，宏内核就是操作系统是个“大管家”，几乎包办一切，用户应用程序的需求直接向内核提出就行；微内核更像一个“代理人”，几乎所有的驱动、文件系统全部运行在与用户应用程序同级的用户态下。

只从便于开发的角度而言，微内核架构对于开发人员更加友好，各服务分离的设计思想也更符合软件工程的理念，这也与目前各个互联网企业中很热门的“微服务”、“高内聚”、“低耦合”的潮流很相近。虽然在宏内核架构当中，也同微内核架构一样管理着 CPU 调度，内存管理，文件管理和系统调用等各模块的的工作，由于用户服务和内核服务被实现在同一空间中，在执行速度上的确要比微内核快。但是，牺牲部分的执行速度换取更稳定的工作与更低的开发难度显然是值得的。

表 1 微内核与宏内核对比

	微内核	宏内核
设计思想	用户服务与内核服务运行空间**不同**	用户服务与内核服务运行空间**相同**
尺寸	较小	较大
执行速度	慢	快
可扩展性	容易扩展	不易扩展
耦合度	低	高
安全性	单个服务崩溃不影响全局	单个服务崩溃意味着整个系统崩溃

题目 2：虚拟化中的权限控制

1.1 各种状态的划分原因

在具体讨论各种状态之前，需要先理解区分各种状态的原因。

为了使程序更快地运行，最“简单粗暴”想法是直接在 CPU 上运行程序。如果按照这种思路设计，一个程序的生命周期应当如下：

当 OS 希望启动程序运行时，它会在进程列表中为其创建一个进程条目，为其分配一些内存，将程序代码加载到内存中，找到 main 函数等入口点，跳转到那里，并开始运行用户的代码。而当这个进程运行结束时，操作系统会将其内存释放，并且从进程列表中清除。

然而，这种“简单粗暴”的方法会在虚拟化 CPU 时产生问题。如果直接在 CPU 上运行程序，操作系统怎么能确保程序不做访问受限资源等危险操作，同时仍然高效地运行？例如，如果希望构建一个在授予文件访问权限前检查权限的文件系统，就不能简单地让任何用户进程向磁盘发出 I/O。如果这样做，一个进程就可以读取或写入整个磁盘，这样所有的针对于磁盘的保护都会失效。

因此，解决的方法是做出多种状态的划分，在各个状态之间设置明确的权限，并提供状态转换的接口，就可以做到高效运行进程的同时保护受限资源。

1.2 核心态

在内核态下，操作系统可以访问机器的**全部资源**。在此状态下，运行的代码可以根据自身需求执行特权操作，如发出 I/O 请求和执行所有类型的受限指令。

1.3 用户态

在用户态下，应用程序**不能完全访问**硬件资源，在用户态下运行的代码将受到限制。例如，在用户态下运行时，进程不能发出 I/O 请求。如果强行这样做会导致处理器引发异常，操作系统可能会终止进程。

通常来说，以下三种情况会导致用户态到内核态的切换

1. **系统调用**
2. **异常**
3. **外围设备的中断**

这三种方式是系统在运行时由用户态转到内核态的最主要方式，其中系统调用可以认为是用户进程主动发起的，异常和外围设备中断则是被动的。也正因为有了核心态与用户态的区分，特权指令与普通指令的区分才有意义。

1.4 特权指令

指令是应用程序向系统发出的命令，操作系统中的指令分为两类——特权指令和非特权指令。

特权指令只能在**核心态**下运行。如果特权指令试图在用户模式下执行，则该指令将被忽略并被视为非法

指令。在执行特权指令时，操作系统可以确保在将控制权转移到任何用户应用程序之前将定时器设置为中断。因此，如果计时器被中断，操作系统可以重新获得控制权。典型的特权指令有清理内存指令、上下文切换指令等等。

1.5 普通指令

普通指令是仅仅在**用户模式**下执行的指令，常用的特权指令有读取系统时间、进行算数运算等等。而最重要的普通指令当之无愧是 **trap** 指令，trap 指令是用户态转变为核心态的重要接口。

关于特权指令与普通指令，需要特别区分的是：

- 直接切换到内核模式的指令是特权指令，因为该指令是独立调用的，而不是通过系统调用、中断或外围设备中断，即之前提到的三种方式。
- 切换到内核模式的系统调用指令是非特权指令，因为该指令在通过系统调用时，执行了 trap 指令。

表 2 特权指令与普通指令对比

	特权指令	普通指令
运行环境	内核态	用户态
限制	用于敏感操作	不共享资源
	在限制下执行	不干扰其他任务
实例	I/O 指令、清除内存	trap 指令、读取系统时间

总而言之，二者最重要的区别为：特权指令是仅仅在核心态下执行的指令，普通指令是仅仅在用户模式下执行的指令。在用户态切换到核心态的三种方式中，最为重要的是**系统调用**。

2.1 系统调用

系统调用是程序发出的由用户态切换到内核态的请求。它允许内核谨慎的向用户程序暴露某些关键功能，例如访问文件系统、创建和销毁进程、与其他进程通信，以及分配更多内存等等。具体的系统调用如题目 4 中提到的 fork()、wait()等等。

系统调用是用户态与内核态之间的重要接口。从底层角度看，能进入内核态的原因是各种系统调用都执行了 **trap** 指令与 **return-from-trap** 指令。要执行系统调用，程序必须执行 trap 指令。该指令同时跳入内核并将特权级别提升到内核态。一旦进入内核，系统就可以执行任何需要的特权操作，从而为进程执行所需的工作。而当系统调用结束后，操作系统执行 return-from-trap 指令，该指令返回到用户程序中，同时将特权级别降低，由内核态回到用户态。

既然 trap 指令作为由用户态切换到内核态的接口，并且所有的系统调用都需要执行 trap 指令，那么对于不同的系统调用而言，trap 指令如何知道在操作系统中应当运行哪些代码呢？

显然，发起系统调用的进程不能指定要跳转到的地址，这样做使得程序可以跳转到内核中的任意位置。而真正的解决方法是设置 **trap table**。

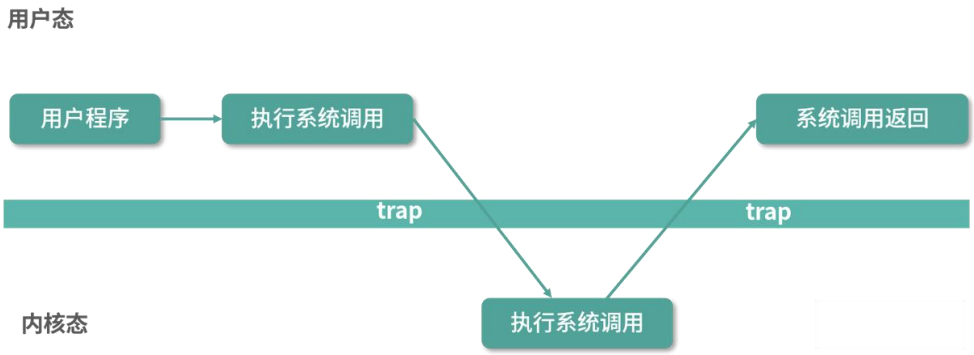

图 3 系统调用执行过程

硬件通过提供不同的执行状态来协助操作系统，还提供了 trap 指令来进入内核态和 return-from-trap 指令从内核态回到用户态，并且提供了一些指令，让操作系统能够告诉硬件 trap table 在内存中的位置。内核在启动时设置 trap table，由于操作系统在内核态下运行，因此可以根据需要自由配置机器硬件。操作系统此时告诉硬件在发生某些异常事件时要运行哪些代码，例如，发生异常、外围设备中断等等。

2.2 库函数

库函数是各类编程库提供的函数。在使用特定的库调用时，首先要导入相关的库，进行调用时对应的库函数就连接到了程序之中。在 C 语言中，程序员可以通过在程序中包含头文件来调用库函数，使用预处理器指令#include 即可导入。

2.3 系统调用与库函数的区别

系统调用与库函数的主要区别在于系统调用是对**内核访问资源**的请求，其中涉及到执行 trap 指令进入内核态；而库调用是使用编程库中定义的**函数的请求**。

首先，man 文档对于系统调用的函数与库函数进行了明确的分类，将系统调用函数放到了第 2 章，而将库函数放入了第 3 章。

```
The table below shows the section numbers of the manual followed by the types of pages they contain.

1   Executable programs or shell commands
2   System calls (functions provided by the kernel)
3   Library calls (functions within program libraries)
4   Special files (usually found in /dev)
5   File formats and conventions, e.g. /etc/passwd
6   Games
7   Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7), man-pages(7)
8   System administration commands (usually only for root)
9   Kernel routines [Non standard]
```

图 4 man 文档对于不同函数类型的区分

然而，系统调用例如 fork()、wait()等等看起来完全就像 C 语言中的库函数，系统如何区分这二者呢?

尽管 fork()、wait()等与 C 语言库函数形式相似，但在执行这些系统调用函数时，会**执行 trap 指令**进入内核态之中。其中，无论是对于 fork()还是提供的其他系统调用，都使用与内核一致的调用约定来将参数、系统调用号放入约定好的栈或寄存器中，之后执行 trap 指令。在 trap 指令之后的代码准备好返回值，将控制权返回给发出系统调用的程序。

题目 3：进程调度与死锁

1.1 MLFQ 算法的必要组成部分

多级反馈队列进程调度算法，即 MLFQ 算法的规则如下：

1. 设置多个优先级队列，队列优先级从高到低递减；不同优先级队列时间片长度不同，长度由高优先级到低优先级递增。

2. 相同优先级队列中的进程采用 FCFS 算法，在时间片内若完成则撤离系统；若未完成则降入下一级队列末尾。

3. 新进程进入系统时，进入最高优先级（最上层队列）的末尾。

4. 当高优先级队列为空时，才可以运行低优先级队列。

其中，第一条规则对于**时间片的长度**的设置有利于各种类型的进程：

- 高优先级队列中，交互型进程较多，较短的时间片可以帮助更快的切换。
- 低优先级队列中，CPU 密集型进程较多，较长的时间片帮助运算。

1.2 MLFQ 算法的设计思想

多级反馈队列进程调度算法（MLFQ）是对于时间片轮转算法和优先级调度算法的综合优化，它主要针对于两方面的问题：

- **周转时间**
- **响应时间**

对于响应时间而言，MLFQ 算法本质上是对时间片轮转算法的一种改进，根据规则，工作进入系统时默认放入最上层队列，可以保证新加入的进程获得比较好的响应时间。因此，无须担心响应时间这一指标。

那么，MLFQ 算法的核心核心问题在于：**没有工作长度这一先验知识，该如何设计调度程序来减少周转时间?**

在基础的最短作业优先算法（SJF）中，算法的前提是已经知道了哪些作业是长度较短的，因此可以让预计用时少的进程先运行，来减少周转时间。这对于抢占式的 SJF 算法更为明显，如果在某个作业正在运行的过程中出现了新作业，如果我们不能对新作业与正在运行作业的长度进行比较，那就不清楚作业的预期运行时间，SJF 算法的必要前提就消失了。而对于真实的复杂环境而言，作业长度是未知的，没有办法满足 SJF 算法运行的这个前提。

因此，MLFQ 的设计思路如下：

首先，要优化周转时间，这通过先执行短工作来实现。然而，操作系统通常不知道工作要运行多久，而这又是 SJF（抢占式或非抢占式）算法所必需的。其次，MLFQ 希望给使用等待交互型进程的用户很好的交互体验，因此需要降低响应时间。然而，像时间片轮转这样的算法虽然降低了响应时间，周转时间却很差。

所以 MLFQ 算法给出的解决办法是希望综合时间片轮转算法与 SJF 算法，同优先级队列内时间片轮转，获得较好的响应时间；新进程到来放入高优先级队列，期望在进程优先级递减的过程中将短进程运行结束。

1.3 具体实例分析

此处的例子，很好的展示了 MLFQ 算法如何近似 SJF 算法。

有两个作业：A 是一个长时间运行的 CPU 密集型作业，B 是一个运行时间较短的交互型作业。假设 A 执行一段时间后 B 到达。

表 3 实例基本信息

- 灰色为交互型作业 A
  - 运行时间为 40
- 黑色为 CPU 密集型作业 B
  - 运行时间为 160
- 共三级优先级队列
  - Q2、Q1、Q0 优先级递减
  - 时间片长度分别为 10、20、30

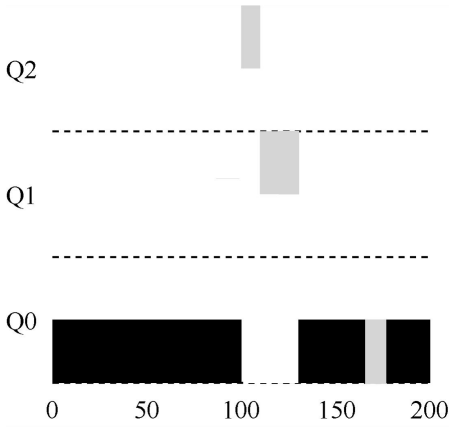

图 5 调度甘特图

A 为长时间运行的 CPU 密集型作业，在最低优先级队列执行（用黑色表示）。B 在时间 T=100 时到达，并被加入最高优先级队列（用灰色表示）。由于它的运行时间只有 40，经过 Q2 与 Q1 队列中的两个时间片，仍未运行结束，进入到 Q0 队列时排在队尾。此时立刻运行 B 进程，这样就很好的解决了**“饥饿”**问题，B 进程使用完分配到的时间片后，继续运行 A 进程。

表 4 调度过程中各级队列与 CPU 情况

时间	0	50	100	110	120	130	140	150	160	170	180	190	200
Q2（时间片长 10）				B									
Q1（时间片长 20）					B	B							
Q0（时间片长 30）	A	A	A				A	A	A	B	A	A	A
CPU 占用	A	A	A	B	B	B	A	A	A	B	A	A	A

表 5 MLFQ 调度算法性能

作业号	提交时间	运行时间	开始时间	响应时间	完成时间	周转时间	带权周转时间
A	100	40	100	0	180	80	0.5
B	0	160	0	0	200	200	0.8

通过这个例子可以看出，MLFQ 算法的一个主要目标：如果不知道作业是短作业还是长作业，那么就在开始的时候假设其是短作业，并赋予最高优先级。如果确实是短作业，则很快会执行完毕，否则将被慢慢移入低优先级队列，而这时该作业也被认为是长作业了。通过这种方式，MLFQ 算法近似于 SJF 算法，从而使得整体获得比较好的周转时间；并且 MLFQ 算法具有时间片轮转的特性，在响应时间上也有很大的优势。

1.4 真实环境下的调度

在真实的操作系统环境之下，Linux 内核有一套高效复杂的调度机制，能使效率极大化，但有时为了实现特定的需求，需要一定的人工干预。例如，希望操作系统能分配更多的 CPU 资源给浏览器进程，让浏览网页速度更快、更流畅，操作体验更好，这就需要手动设置优先级。在 Linux 中，命令行工具 nice 可以增加或者降低作业的优先级，从而增加或降低相应进程在某个时刻运行的机会。下图为 nice 工具的 tldr 文档，简易介绍了使用方式。

```
❯ tldr nice

  nice

  Execute a program with a custom scheduling priority (niceness).
  Niceness values range from -20 (the highest priority) to 19 (the lowest).
  More information: https://www.gnu.org/software/coreutils/nice.

  - Launch a program with altered priority:
    nice -n niceness_value command
```

图 6 nice 工具的 tldr 文档

2.1 死锁的概念

死锁是指两个或两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程。

一个常见的例子：如果某一个计算机系统只存在一台打印机与一台输入设备，若此时进程 P1 占用了输入设备，同时提出了使用打印机的请求，但此时打印机正在被进程 P2 占用。如果在 P2 进程没有释放掉打印机之前，又提出请求申请正在被 P1 占用的输入设备。此时 P1、P2 进程会相互无休止等待下去，均无法继续执行。这样，这两个进程就陷入了死锁。

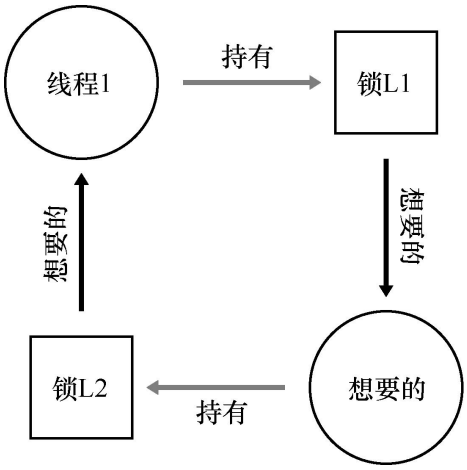

图 7 死锁示意图

2.2 死锁产生的必要条件

- **互斥**：进程对于需要的资源进行互斥的访问（例如一个进程抢到锁）。
- **持有并等待**：进程持有了资源（例如已经持有的锁），同时又在等待其他资源（例如需要获得的锁）。
- **非抢占**：进程获得的资源（例如锁），不能被抢占。
- **循环等待**：进程之间存在一个环路，环路上每个进程都额外持有一个资源，而这个资源又是下一个进程要申请的。

死锁的产生必须同时具备以上四个必要条件，如果上述四个条件之一不具备，那么死锁将不会产生。所以若想要使得系统避免发生死锁，破坏四个条件之一即可。

2.3 死锁安全状态与安全序列的关系

安全状态是指系统能按某种进程推进顺序(P1,P2,…,Pn)为每个进程 Pi 分配其所需资源，直至满足每个进程对资源的最大需求，使每个进程都可顺序地完成。

而安全序列是指，如果对于每一个进程 Pi(1≤i≤n)，一个进程序列{P1,P2,…,Pn}是安全的，它以后还需要的资源量不超过系统当前剩余资源量与所有进程 Pj(j<i)当前占有资源量之和。这样的序列即为安全序列。

并非所有的不安全状态都是死锁状态，但当系统进入不安全状态后，才可能进入死锁状态；相反，只要系统处于安全状态，系统便可以避免进入死锁状态。即系统进入不安全状态是死锁的**必要条件。**

2.4 银行家算法

2.4.1 算法数据结构

- **Avaliable[m]**--可利用资源向量。含有 m 个元素的数组，Available[j]=k 表示系统中第 j 类资源数为 k 个，其初值是系统中所配置的该类全部可用资源数目，它的值随该类资源的分配和回收而动态地改变。
- **Max[n,m]**--最大需求矩阵。n*m 的矩阵，它定义了系统中 n 个进程中的每一个进程对 m 类资源的最大需求。Max[i,j]=k 表示进程 i 对 j 类资源的最大需求数为 k。
- **Allocation[n,m]**--分配矩阵。n*m 的矩阵，它定义了系统中每一类资源当前已分配给每一个进程的资源数，Allocation[i,j]=k 表示进程 i 已分得 j 类资源的数目为 k 个。
- **Need[n，m]**--需求矩阵。n*m 的矩阵，它定义了当前系统的 n 个进程要想完成工作，还需要各类资源的数目，它表示每个进程尚需的各类资源数，Need[i,j]=k 表示进程 i 还需要 j 类资源 k 个。

上述的三个矩阵存在以下关系：

```
Need[i,j] = Max[i,j] - Allocation[i,j]
```

在银行家算法中，默认 Need 矩阵为已知条件，而这在现实情况中并不成立，详细分析过程在银行家算法适用条件部分会展开分析。

- **Request**--进程 Pi 的请求向量 Request。一个含有 m 个元素的向量，它表示进程 i 对各类资源的需求情况。如果 Request[j]=k，表示进程 Pi 需要 k 个 j 类型的资源。

安全性检查算法也是银行家算法的步骤之一，其中需要用到两个向量：

- **Work**--表示系统可提供给进程继续运行所需的各类资源数目，它含有 m 个元素
- **Finish**--表示系统是否有足够的资源分配给进程，使之运行完成。开始时 Finish[i]:=False，当有足够资源分配给进程时，再令 Finish[i]:=True。

2.4.2 算法过程

假设此时进程 Pi 提出资源请求 Request[j]=k；

1. 若 Request[i]<=Need[i,j]便转向执行步骤 2；否则认为出错。
2. 若 Request[i]<=Available[i,j]便转向执行步骤 3；否则表示系统尚无足够的资源分配，让 Pi 等待。
3. 系统假设将 Pi 所要求的资源分配给 Pi，并对数据结构做如下修改：

```
Avaliable[j] =Available[j] - Request[j]
Allocation[i,j] = Allocation[i,j] - Request[j]
Need[i,j] = Need[i,j]- Request[j]
```

4. 系统执行**安全性算法**，检测此次资源分配后，系统是否处于安全状态。若安全才正式将资源分配给进程 Pi，以完成本次分配；否则，本次试探分配作废，恢复原来的资源分配状态，让 Pi 等待。

- **安全性算法**步骤如下：

1. 在执行安全算法开始时，分别设置两个向量：

```
Work:=Available; Finish [i] := False;
```

2. 从进程集合中找到一个能满足以下条件的进程：

```
Finish[i] = false;
Need[i,j] <= Work[j];
```

3. 进程 Pi 获得资源后，可以顺利执行，直至完成，并释放分配给它们的资源，故应执行：

```
Work[j] = Work[j] +Allocation[i,j];
Finish[i] = true;
```

此时转向步骤 2

4. 若此时安全序列中已经包含了所有进程，则表示系统处于安全状态；否则系统处于不安全状态。

由此可见，**安全性算法实际是银行家算法的一部分**，为当前系统找到一个安全序列，确保此刻的系统的分配状态能够保证内存中的进程全部顺利完成。

2.4.3 算法适用条件

银行家算法本质上属于死锁避免中的一种，通过系统调度防止系统进入不安全状态。然而类似于银行家算法的调度算法在避免死锁时有很大的局限性，因此通过调度来避免死锁并不是广泛的解决方案。

1. 类似于在介绍 MFLQ 算法时提到的 SJF 最短作业优先算法需要有作业预期运行时间的先验知识，此处的银行家算法也需要必须有所有进程最大资源需求的**先验知识**。

2. 要求进程数固定，它正在执行时不能启动其他进程。这一点使得银行家算法有一种“**静态**”的感觉，当银行家算法运行的那一刻，好像按下了照片的快门进行了定格，此后的过程必须严格的按照银行家算法执行。在真实的操作系统中，这一点很明显是不可能做到的。

3. 要求资源数量保持固定。在没有发生死锁的可能性的情况下，任何资源都不得因任何原因而下降。

4. 没有考虑进程的运行时间。它允许在有限时间内批准所有请求，但是一秒钟，一天，一个月都是有限的时间，这显然是不合理的。

题目 4：哲学家进餐问题

1.1 问题描述

哲学家进餐问题是一个经典的并发问题，它由 Dijkstra 提出来并解决。

假定有 5 位哲学家围着一个圆桌，每两位哲学家之间有一把叉子（一共 5 把）。哲学家有时要思考一会，不需要叉子；有时又要就餐。而每位哲学家只有同时拿到了左手边和右手边的两把叉子，才能吃到东西，涉及到了叉子的竞争以及随之而来的同步问题。

功利一点来说，哲学家进餐问题是就业或者读研面试是经常要问到的部分，尤其是工作时的后端开发岗位，这也是著名的“面试八股”题目之一。

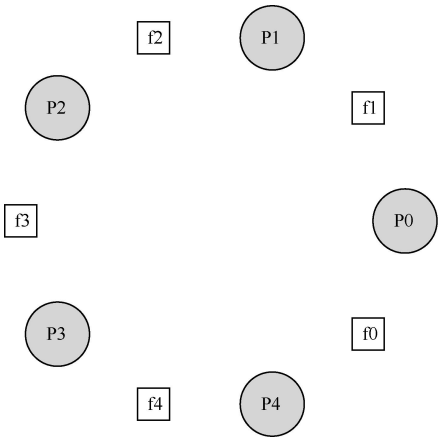

图 8 哲学家进餐问题示意图

1.2 C 语言实现

1.2.1 PV 操作

在解决哲学家问题之前，首先要熟悉信号量机制在 PV 操作中的运用。

PV 操作均为**原子操作**，原子操作不会被进程调度打断，这种操作一旦开始就会一直运行到结束，而不会有任何的上下文切换。这也就保证了 S 的值在机器将其翻译成汇编语句，在寄存器上进行加 1、减 1 操作时是连贯的。

使用 **P 操作**，即 wait 申请资源当信号量的值 S.value < 0 时，表示该资源已经分配完毕，需要调用 block() 原语使进程从运行态变为阻塞态，并把当前进程挂到信号量 S 的等待队列当中。

```
void wait(semaphore S) {
    S.value --;
    if (S.value < 0) {
        block(S.L);
    }
}
```

```
void signal(semaphore S) {
    S.value ++;
    if (S.value <= 0) {
        wake(process p);
    }
}
```

图 9 P 操作（图左）V 操作（图右）

而释放资源需要用到 **V 操作**，当 S.value <= 0 时，表示当前仍有等待该资源的进程被阻塞在信号量 S 的等待队列当中，调用 wake()原语将 S.L 中的第一个进程从阻塞态变为就绪态。

1.2.2 互斥问题

1. 设置互斥信号量 mutex，并设初值为 1。
2. 在访问临界资源之前，执行 p(mutex)，
   把 mutex 的值-1，表示当前有进程正在访问临界资源。
3. 在结束访问之后，执行 v(mutex)，
   把 mutex 的值+1，表示访问临界资源的进程访问结束。

```
semaphore mutex = 1;

P1() {
    p(mutex);
    访问临界资源;
    v(mutex);
}
```

图 10 PV 操作实现互斥

信号量是有一个整数值的对象，可以用两个函数来操作它。在 POSIX 标准中，是 sem_wait()和 sem_post()，即分别对应着 P 操作与 V 操作。对于哲学家问题的具体解决而言，为了调用方便，可以使用 C 语言的 define 语句直接将 POSIX 提供的对应函数定义为 P、V 操作。在 main 函数中，针对 5 个哲学家分别创建了 5 个线程。叉子为临界资源，在初始化时都设置相应信号量值为 1，并且在最后使用 pthread_join 语句将分配出去的 5 个线程资源进行回收。

```
#include <pthread.h>
#include <semaphore.h>

#define P sem_wait
#define V sem_post

sem_t forks[5];
pthread_t p[5];

int main(int argc, char *argv[])
{
    int i;
    for (i = 0; i < 5; i++)
        sem_init(&forks[i], 0, 1);

    for (i = 0; i < 5; i++)
        pthread_create(&p[i], NULL, philosopher,
&i);
    for (i = 0; i < 5; i++)
        pthread_join(p[i], NULL);
    return 0;
}
```

图 11 宏定义与线程创建

对于每一个哲学家而言，他们的基本循环如下图所示，他们进行思考--取叉子--吃饭--放叉子的过程。

```
void *philosopher(void *arg)
{
    int id = *((int *)arg);
    think();
    get_forks(id);
    eat();
    put_forks(id);
    return NULL;
}
```

图 12 哲学家线程具体工作流程

解决哲学家进餐问题关键在于实现 **getforks()**和 **putforks()**函数，保证没有死锁，没有哲学家会饥饿，并且并发度更高。

一种错误的思路是，为了拿到叉子，依次访问每把叉子——先左手边的，然后是右手边的。结束就餐时再将叉子释放掉。虽然如此操作可以实现对于叉子的互斥访问，但是这样的操作会导致死锁的产生。

假设每个哲学家都拿到了左手边的叉子，此时阻塞就会发生，并且都在一直等待另一个叉子。具体来说，当 0、1、2、3、4 五位哲学家分别拿到了序号对应的 0、1、2、3、4 五只叉子时，此时所有的餐叉都被占有了，所有的哲学家都阻塞着，并且等待另一个哲学家释放掉占有的叉子。

```c
void get_forks(int p)
{
    if (p == 4)
    {
        P(&forks[right(p)]);
        P(&forks[left(p)]);
    }
    else
    {
        P(&forks[left(p)]);
        P(&forks[right(p)]);
    }
}

int left(int p)
{
    return p;
}

int right(int p)
{
    return (p + 1) % 5;
}
```

```c
void put_forks(int p)
{
    V(&forks[left(p)]);
    V(&forks[right(p)]);
}

void think()
{
    return;
}

void eat()
{
    return;
}
```

图 13 取叉子（图左）放叉子（图右）具体实现

解决上述问题最简单的方法，就是修改某个哲学家的**取叉子顺序**。此处改变了 4 号哲学家的策略，不同于其他哲学家，4 号哲学家先拿右手边的叉子，后拿左手边的叉子，这样就不会出现每个哲学家都拿着一个叉子，卡住等待另一个的情况，此时破坏了死锁必要条件之一，循环等待。

程序运行结果如右图所示：

3 号哲学家拿到 fork 3 之后，fork 4 被 4 号哲学家先取走，此时 3 号哲学家等待 4 号哲学家进餐完毕后才可以取到 fork 4，完成进餐。程序运行结果正确。

代码在此题中的之前介绍的代码片段的基础上，使用 PV 操作，添加了对于 **stdout 标准输出**的互斥访问，使得同时只可以有 1 个哲学家线程访问 stdout 标准输出这一临界资源。

```
❯ ./phi_dinning 1
started

0: start
0: think
get fork 0
get fork 1
0: eat
0: done
----------------------------------------4: start
----------------------------------------4: think
----------------------------------------get fork 0
--------------------2: start
--------------------2: think
--------------------get fork 2
--------------------get fork 3
--------------------2: eat
--------------------2: done
------------------------------3: start
------------------------------3: think
------------------------------get fork 3
----------1: start
----------1: think
----------get fork 1
----------get fork 2
----------1: eat
----------1: done
----------------------------------------get fork 4
----------------------------------------4: eat
----------------------------------------4: done
------------------------------get fork 4
------------------------------3: eat
------------------------------3: done

finished
```

图 14 哲学家运行顺序

题目 5：系统调用与 main 函数

1.1 fork()

fork 函数通过复制调用它的进程，产生一个新的进程。新产生的进程为子进程，调用 fork 函数的进程是父进程。子进程不会从 main()函数开始执行，而是从调用 fork()的位置返回。

对于 fork 函数而言，当一个进程调用 fork 函数后，通过系统调用的方式创建子进程。系统先为子进程分配资源，例如存储数据和代码的空间，然后把父进程的所有值都复制到新的子进程中。子进程的运行环境是从父进程中 fork 一份资源，拷贝了相同的环境，但由系统创建 PCB 时给予一个新的 pid。此处的“拷贝”大多数情况可以理解为编程语言中说的“**深拷贝**”，即把相关变量在子进程的内存中创建，它们的地址值不同于父进程的原始值地址。父进程和子进程运行在不同的地址空间中。在 fork 结束之后，父进程和子进程的内存空间会有相同的内容。对于其中一个进程的内存写入、文件映射并不会影响到另一个进程。

子进程并不是完全拷贝了父进程。具体来说，虽然它拥有自己的地址空间(即拥有自己的私有内存)、寄存器、程序计数器等，但是它从 fork()返回的值是不同的。父进程获得的返回值是新创建子进程的 PID，而子进程获得的返回值是 0。正是因为这样的差别，就很容易为父子进程这两种情况编写不同逻辑的代码。

尽管大部分子进程对于父进程的拷贝是”深拷贝“，但仍有些特例。这里举一个与我们所学内容比较相近的例子，在下学期的《操作系统(2)》涉及到文件管理的时候很有可能会遇到。子进程继承父进程**文件描述符**，这也就意味着，打开父进程打开一个文件之后，fork 一个子进程，父子进程均向这个文件内写入内容。

```c
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<fcntl.h>
#include<string.h>
#include<sys/wait.h>

int main()
{
    int fd = open("./test.txt", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
    int rc = fork();
    if (rc == 0) {
        char * s = "child write something!\n";
        write(fd, s, strlen(s));
    } else {
        char * s = "parent write something\n";
        write(fd, s, strlen(s));
        wait(NULL);
        close(fd);
    }
    return 0;
}
```

图 15 子进程继承父进程文件描述符

运行结果如下图所示，尽管此文件描述符是父进程在 fork 子进程之前创建的，由于子进程继承了父进程的文件描述符，此处的继承是“**浅拷贝**”，因此父子进程都可以对同一个文件进行写操作。

```
1  parent write something
2  child write something!
3
```

图 16 子进程继承父进程文件描述符

关于 fork 的介绍此处再提一个我个人认为有趣的点，尽管它与系统调用的知识点无关。在系统调用中一般叫做是父进程 fork 出子进程，其中的主语是“父进程”；而 Github 上一般说 fork 别人的仓库，这个主语是“子仓库”。

1.2 exec()

首先纠正一点题干中叙述的不严谨，准确的说，exec 族函数并非系统调用，execve 函数才应当叫做系统调用。在回答第二题时曾经也提到过，man 文档中第二个部分 man 的 2 号章节提供的是内核提供的系统调用，3 号章节提供的是由库提供的函数。正如下图中 exec 族函数 execl、execlp 等等属于 man 文档的 3 号章节，而 execve 函数属于 man 文档的 2 号章节。

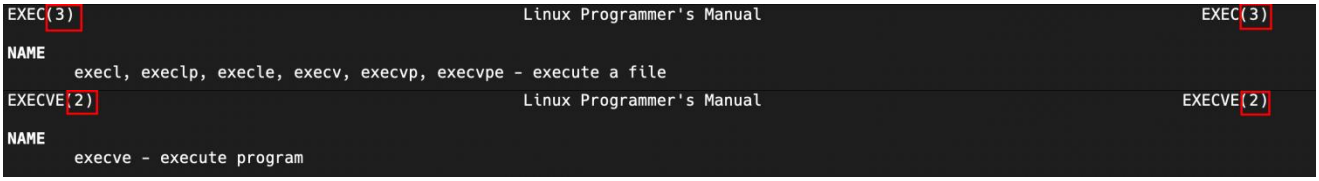

图 17 man 文档中对于 exec 族函数与 execve 函数的不同章节定义

根据 man 文档，exec 族函数，例如 execl、execlp 等等，其实都是 execve 函数的前端，即调用这些 exec 函数实际上是调用了 execve 函数，只不过是 exec 族函数帮助封装了各种参数，方便我们的调用。exec 族函数包含以下几种变体：execl、execle、execlp、execv、execvp，通过一个新的进程镜像来替代当前函数，可以让子进程与父进程运行不同的程序。

exec()会从可执行程序中加载代码和静态数据，并用它覆写自己的代码段等静态数据，堆、栈及其他内存空间也会被重新初始化。然后操作系统就执行该程序，将参数通过 argv 传递给该进程。因此，它并没有创建新进程，而是直接将当前运行的程序替换为此处使用 exec 运行的程序，所以调用 exec 族函数前后该进程的 PID 并不改变。这样的执行逻辑与直接在 shell 中执行命令是不一样的，在 shell 中直接执行命令会新建一个进程运行此程序。

而 exec 族函数的后端 execve 函数与 main 函数有很大的联系，此部分介绍放到下一题中具体讨论

1.3 wait()

父进程调用 wait 系统调用来等待并监听子进程状态的改变。根据 man 文档，这里所说的的状态改变有多种情况，具体为：

- 子进程终止
- 子进程被信号暂停
- 子进程被信号恢复

父进程暂停系统的执行，直到子进程之一终止，wait 系统调用会在子进程运行结束后才会返回。通过这样的操作，即使父进程本身应当先运行结束，也会等到子进程结束后再 wait()返回。调用 wait 系统调用成功时，wait 返回值为停止的子进程的 pid，失败时，返回值为-1。

本质上，执行 wait(&wstatus)等同于执行 waitpid(-1, &wstatus, 0)。而我们在之前做上机实验时，对于 wait

的功能只用到了等待**子进程终止**这一种情况，即 wstatus 参数设置为 NULL，调用 wait(NULL)，默认使得父进程等待子进程的终止。

对于被终止的子进程而言，调用 wait 系统调用可以使得系统释放掉与子进程相关联的资源；而不调用 wait 系统调用的话，如果父进程比子进程先执行结束，那么子进程就成为了孤儿进程。孤儿进程将被 init 进程（进程号为 1）所收养，并由 init 进程对它们完成状态收集工作。对于 init 进程，他循环地进行 wait 操作，从而保证子进程的结束。

1.4 exit()

exit 系统调用会通过向进程发送 SIGCHLD 信号的方式，立即停止调用它的进程。所有属于这个进程的文件描述符都将被关闭，这个进程的所有的子进程都将被的 init 进程所继承。

1.5 getpid()

getpid 系统调用会返还调用这个函数的进程的 pid。

pid 是进程标识符，每个进程都有唯一的 pid，它在进程运行时由系统分配，存储在 PCB 的内部。在进程运行时，PID 是并不会改变的，但是进程终止后 PID 标识符就会被系统回收，就可能会被继续分配给新运行的程序。

2.1 main 函数完整格式

main 函数完整格式如下图所示：

```
int main(int argc, char *argv[], char *envp[])
```

其中的参数需要详细解释，便于理解 main 函数与 execve 系统调用的的关系。argc 为 argument count，argv 为 argument vector，分别是参数个数，参数数组。此处的 argv 是一个指向有 argc + 1 个 char 类型数组的指针，是一个指向数组的指针。它指向的的位置是含有 argc + 1 个指针的指针数组，这个指针数组指向 char 类型的参数，且这个指针数组的第一个指针 argv[0]指向这个 C 文件本身的文件名，最后一个指针 argv[argc]指向 null。这也解释了 argv 指向的长度为什么是 argc + 1 个，而 argc 最小值为 1。即如果并不向 C 最终的二进制文件传入参数的话，唯一的一个参数是文件名称本身。

有趣的是，argv 数组最后一个虽然指向为空，如果人为的制造指针指向偏移，可以发现 argv 之后的指针指向正是环境变量。使用下面的程序对参数进行打印验证，第三个参数 envp 指向的是环境变量指针数组。

```c
#include <stdio.h>

int main(int argc, char *argv[], char **envp)
{
    printf("argc: %d\n", argc);
    for (int i = 0; i <= argc + 3; i++)
    {
        printf("argv[%d]: %s\n", i, *(argv + i));
    }
    return 0;
}
```

图 18 输出打印 argc argv envp

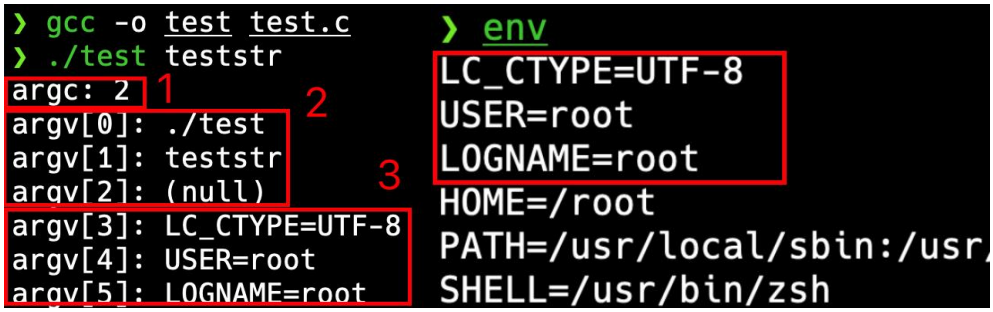

图 19 输出打印结果（左） 对应的环境变量（右）

观察上图结果，当执行./test teststr 时，上图（左）编号 1 处可看到 argc 值为 2，即./test 与 teststr 两个参数，根据之前的分析可明确 argv 数组指向长度为 argc+1，此处为 3，并且 argv[0]指向文件名./test，最后一个指针 argv[2]指向空指针。当我们人为制造指针地址偏移时（此处为多偏移了三个单位），可以看到指针指向了各种环境变量，例如我的文件默认编码格式 UTF-8，用户名 root，登录名 root 等等。上图（右）在 shell 中输入 env 命令，截取了部分环境变量，也可以看出前三行相对应。

2.2 main 函数入口参数与 exec 关系

首先，正如之前在介绍 exec 族函数时所说，exec 族函数共六个，均为 execve 函数的前端，即 exec 族函数调用后最终结果均为调用 execve 函数，execve 函数原型与 main 函数原型如下。

```
int execve(const char *filename, char \*const argv[], **char \*const envp[]);
int main(int argc, char *argv[], char *envp[]
```

与 main 函数的函数参数相同，在调用 execve 函数时，argv[0]应该指向要执行的文件名称，envp 指向环境变量，用键值对的形式存储，具体写法可以参考介绍 main 函数部分时的图 19，输入 shell 命令 env 后的输出内容。需要注意的是，argv 与 envp 数组最后一定都要指向空指针。

因此，当调用 exec 族中六个函数中的一个时，他们会作为 execve 函数的前端，再次调用 execve 函数。execve 函数参数中的的 argv、envp 两个参数即可以传递给文件名为 filename 的那个程序的 main 函数，这也就完成了执行其他程序这一功能。

2.3 main 函数返回值与系统调用 exit 的关系

exit 与 C 语言的 return 语句有关。exit 函数的函数原型为 void exit(int status)。C 语言在 return 的值如果可以类型转换为 int，就和执行系统调用 exit()相同，并且把 return 的那个值作为参数返还给操作系统。exit(0)表示正常退出，exit(x)（x 不为 0）都表示异常退出。

若使用返回语句，则返回值会用作隐式调用 exit()的参数。值零和 EXIT_SUCCESS 指示成功终止，值 EXIT_FAILURE 指示不成功终止。

2.4 main 函数入口参数与返回值和 shell 语言内置变量的关系

main 函数的入口参数中的 argv 实际上与 shell 语言内置变量$1、$2 等等一一对应，其中的 1、2 即为第 1

个参数、第 2 个参数等等。而 main 函数的返回值也与 shell 内置变量$?对应。

下面运行了一个测试的 C 程序，这个 C 程序其最终执行 return 0。$?记录了上一条执行后的命令的返回值，使用 shell 的$查看上一次执行结果的返回值可知为 0，即 return 本质调用了 exit(0)，与介绍 exit 的部分呼应。此处的成功返回也可以从打红框绿色箭头看出来（图上），绿色箭头是我使用的 zsh 提供的展示返回值的功能。

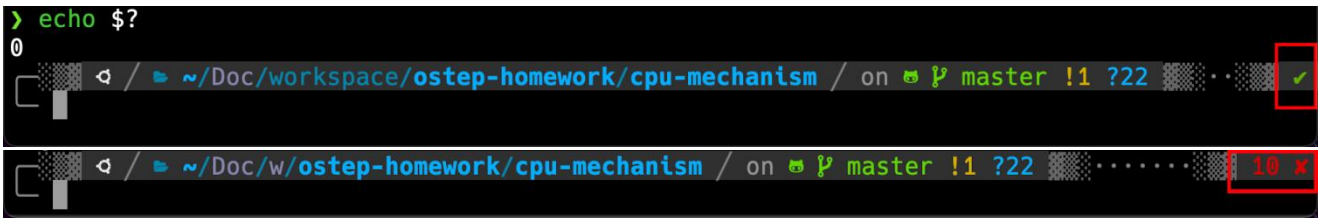

图 20 返回值为 0（图上）返回值为 10（图下）

而当我修改 C 程序，将其 return 值改为 10 时，可以看到我的 zsh 提示返回值也变为了 10（图下），由于 0 为成功返回，10 返回值异常，因此会有 X 的错误提示，如（图下）所示。

3.1 多分支进程树

上机时最让我印象深刻的程序为创建进程树的程序，此程序创建出了进程家族树，很好的展示了父子进程之间的关系。

```c
/*建立进程树*/
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>

int main()
{
    int i;
    printf("My pid is % d, my father's pid is % d\n", getpid(), getppid());
    for (i = 0; i < 3; i++)
        if (fork() == 0)
            printf("%d pid = % d ppid = % d\n", i, getpid(), getppid());
        else
        {
            int j = wait(0);
            printf("%d: The child % d is finished.\n", getpid(), j);
        }
    return 0;
}
```

图 21 多分支进程树

子进程的运行环境是从父进程中 fork 一份资源，拷贝了相同的环境，但由系统创建 PCB 时给予一个新的 pid。在这个程序中，子进程对于父进程的资源，变量 i 进行了 fork，这个拷贝可以理解为编程语言中说的“深拷贝”。把相关变量在子进程的内存中创建，它们的地址值不同于父进程的原始值地址。所以在子进程中修改 i 不会影响父进程中的 i。

在下图中，pid 为 10511 的进程是执行的 shell 程序，他是运行二进制文件程序的父进程，它的 pid 不会随着多次运行代码而改变。pid 为 13814 进程是负责运行二进制文件的进程，此二进制文件由 C 语言编译后产生获得。反复运行此程序时，虽然 pid 每次各不相同，但是执行结果的顺序与逻辑关系不变。此程序的执行方式也类似于 **DFS**，对于进程家族树进行前序遍历，也体现出进程家族树“树”的数据结构特点。

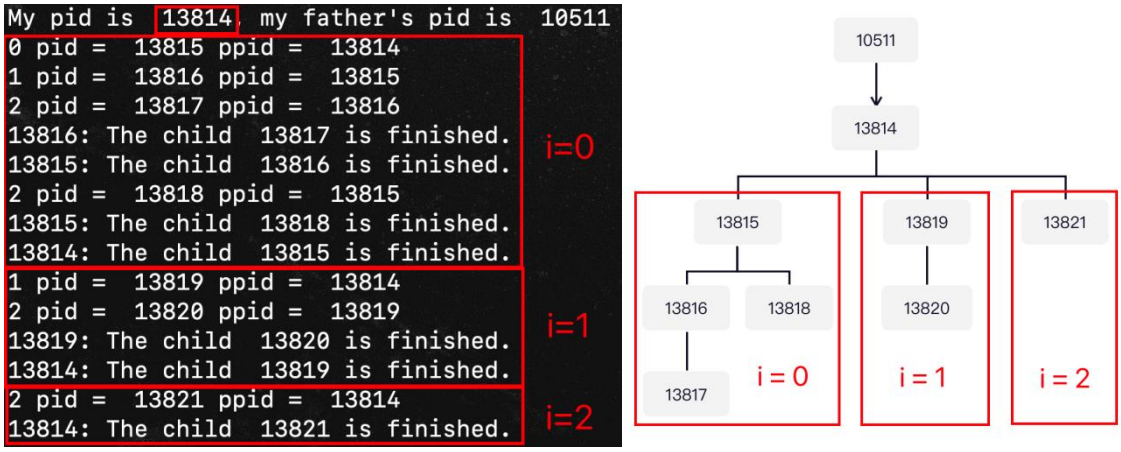

图 22. 父进程在循环中 i 值对应状态（图左） 进程家族树（图右）

3.2 单分支进程树

通过 wait 函数的特性来控制产生单分支进程树，wait 函数的作用之一是监控子进程运行状态，只要有一个子进程运行结束，wait 函数就会返回该子进程的进程号。

因此修改父进程部分的逻辑，当只要 wait 函数监控到有一个子进程运行结束，就退出循环，这样就不会产生新的子进程，也就完成了单分支进程树的要求。核心代码如下图所示：

```
for (i = 0; i < 3; i++){
        if (fork())
        {
            j = wait(0); //等待子进程结束
            printf("%d: The child %d is finished.\n", getpid(), j);
            break;
        } //结束父进程，避免产生新的子进程
        else
        { //进入子进程
            printf("%d pid = % d ppid = % d\n", i, getpid(), getppid());
        }
}
```

图 23. 单分支核心代码

按照上述代码修改，进程的创建与消亡又类似于**栈**的方式，父进程先创建后后终止，子进程后创建先终止。即栈的“**先进后出**”。具体顺序如下图左红色箭头部分。

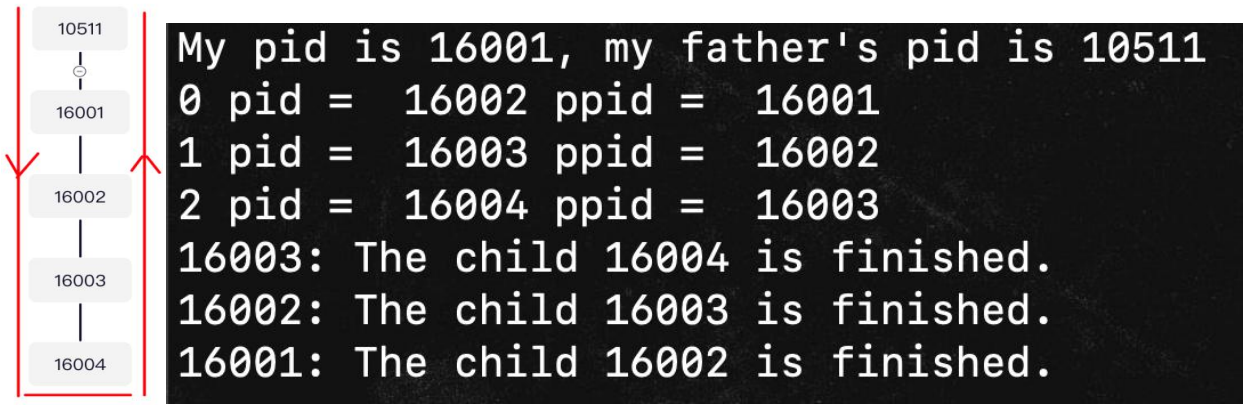

图 24. 单分支结果

值得注意的是，上图中第一行的 my father's pid is 所输出的进程号 10511 仍与多分支进程家族树中相同，原因是这两种方式虽然执行的可执行文件是不同的，但两种情况的父进程都是运行他们程序的 shell，此处没有创建新的 shell，自然 pid 也是不变的。

3.3 "小程序"组合的设计思想

通常，在系统调用 fork 之后，子进程使用 exec 族函数，用新程序来取代进程的内存空间。exec 族函数加载二进制文件到内存中（破坏了包含 exec 族函数的原来程序的内存内容），并开始执行。采用这种方式，这两个进程能相互通信，并能按各自方法运行。父进程能够创建更多子进程，而如果在子进程运行时没有什么可做，那么它采用系统调用 wait 把自己移出就绪队列，直到子进程终止。因为调用 exec 用新程序覆盖了进程的地址空间，所以除非调用 exec 时出现错误，否则不会返回控制。

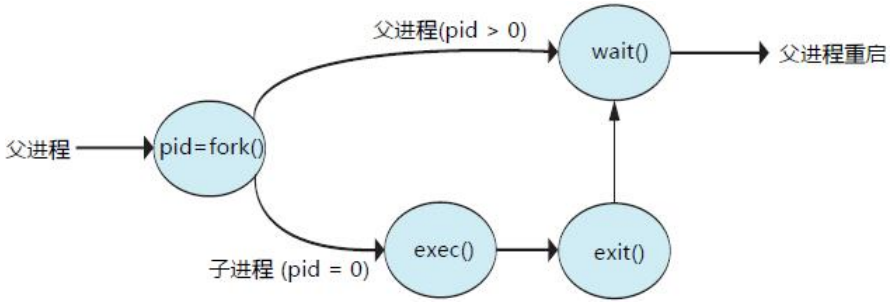

图 25. 父子进程间执行不同程序段

fork() exec() wait() 这一套"组合拳"也和 shell 的设计思想相吻合。shell 本质上也是一个程序，它显示一个命令提示符 prompt，提示用户输入。当用户输入指令之后，shell 可以从文件系统中找到这个可执行程序，调用 fork()创建新进程，并调用 exec()的某个变体来执行这个可执行程序，调用 wait()等待该命令完成。子进程执行结束后，shell 从 wait()返回并再次输出一个提示符，等待用户输入下一条命令。

除此之外，shell 实现结果重定向的方式也很简单，例如下面的代码，完成统计 testfile.c 中的字符，并且重定向输出到 newfile.txt 文件之中。

```
wc testfile.c > newfile.txt
```

这种分离 fork()及 exec()的做法确实更有利于构建 shell，因为这给了 shell 在 fork 之后，exec 之前运行代码的机会，而正是 fork 与 exec 之间"夹着"的这些代码，可以在 exec 运行新程序前改变环境，从而让一系列有趣的功能很容易实现。当完成子进程的创建后，shell 在调用 exec()之前先关闭了标准输出 stdout，打开了文件 newfile.txt。这样，即将运行的程序 wc 的输出结果就被发送到该文件，而不是打印在屏幕上。

当然，fork 这种创建进程的写法似乎完全不同于我们之前所熟悉的方式，不过 fork()、exec()、wait()这样的组合十分强大，从 Linux 设计哲学上说，"小程序集合而成的大型应用程序比单个的大程序更灵活，也更为实用"，这三个系统调用间的配合完美的诠释了这样的思想。

心得与体会

结束了一个学期对于操作系统的学习，完成了操作系统的期末报告，感触颇深。

经过这次期末报告五道题的洗礼，最重要的，就是我真正体会到了 **RTFM** 的重要性。RTFM，是指 Read the Friendly Mannul，这句话通常用在回复那些只要查阅文档就可以解决，但却拿出来提问浪费别人时间的问题。这是一个属于程序员之间的“梗”，往往程序员们谈到 RTFM 时，都会会心一笑。类似的梗还有 **STFW**，即 Search the Friendly Web。毕竟许多问题在自己遇到之前一定也有其他人已经遇到过，官方文档与搜索也一定说明过许多共性的问题。**独立解决问题的能力**，被看作程序员的立业之本。自己顺着文档的思路，亲自动手，对于问题的体会一定更深，收获也会更多。这个感受在完成报告撰写时，回答系统调用的功能简述这个问题时尤为深刻。许多只需要查阅官方文档的简单问题，在相关论坛上被回答的千奇百怪。而 man 手册、c 语言官方文档 cppreference、开源社区维护的工具 TLDR 等等，不仅权威，质量也更高。

在某些质量较差的问答论坛上，很多回答问题的帖子仅仅只是对于别人的回答进行了“fork”，盲目的抄袭。而这种 fork 导致的结果是，如果父帖子的回答出错的话，那所有“fork”出来的子回答所“继承”的资源更可想而知。大家都互相抄，一份本身就质量不高的回答反复被搜索引擎“调度”，占用过多的“时间片”。在这样一个糟糕的技术社区里寻找思路与答案，是对时间的一种浪费。

官方文档基本建立在读者有一定基础水平的设定上编写的，就好像方便面的说明书只会告诉你加水泡多久，并不会告诉你按热水壶上哪个键才可以烧开水。早就听闻过阅读官方文档的重要性，但惭愧的说，在大一大二时也曾尝试过这样的学习方式，不过由于当时基础薄弱，看了没多久就劝退了。对于各种命令只是会调用接口，死记硬背住了格式，从未思考过背后的执行过程，成了名副其实的“调包侠”。不过这次的确把 main 函数、各类系统调用的相关部分的官方文档研读了好几遍。在读懂文档、理解文档的前提下，设计程序对于官方文档提到的内容进行验证，确保真正理解了内部的原理。

对文档读的更加细致，越能体会到作者的思想与智慧。也正是这次期末报告的机会，让我对操作系统的认知与理解不仅仅停留在考试之前刷一周《王道考研 408》，拿一个还算不错的绩点。而几周过去，只是浅浅的留一个印象，卷到了绩点，但是少了必要的思考与训练的过程。做题目，做卷子，只是一种对知识掌握程度的检测方式，但更应该挖掘的知识还是应当亲自去用、去思考、去体味。

在平时完成上机实验时，许多细节的问题的确没有思考。随着知识体系的完备，我感受到了一条条命令背后的执行逻辑原来是如此奇妙，更是对 Linux 设计思想中的天才想法赞叹不已。而也正是一点点揭开 Linux 运行逻辑所带来的成就感，激励我深入的探究下去。