上 海 大 学

2022-2023 秋季学期

《操作系统(1)（08305011）》课程考核报告

学 号：20121802

姓 名：严昕宇

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

一、注意事项：

1、课程考核必须由考生独立完成。考核报告提交结束后将进行查重处理，对有抄袭现象的材料，考核成绩作 0 分处理!!!

2、考核报告在 **11 月 23 日中午 12：00 之前**提交到超星平台上任课老师课程作业的相关目录，逾期没有提交的同学作缺考处理。

3、考核报告用 PDF 文件格式，文件名格式为：学号-姓名.PDF,例如：19120000-张三.PDF。

二、考核题目

题目 1 （20 分）

在操作系统发展史上，微内核是一个十分重要的操作系统体系结构，总的来说微内核并不是一个成功的体系结构，在主流的操作系统中，没有采用或者抛弃了微内核结构，但是微内核结构仍然吸引学术界的关注，也存在一些基于微内核构建的操作系统。请你根据自己的理解回答下列问题：

（1） 什么是微内核？（2 分）微内核中包括哪些基本功能。（4 分）

（2） 和宏内核相比，微内核有哪些优点？（5 分）存在哪些问题？（2 分）

（3） 请查询资料介绍两个微内核（2 分）和两个宏内核（2 分）的实例。

（4） 如果由你设计实现一个操作系统内核，你是采用微内核还是宏内核？给出你的理由。（3 分）

题目 2 （20 分）

请你根据自己的理解回答下列问题：

（1） 请简答核心态、用户态、特权指令、普通指令四个概念（每个 2 分共 8 分），以及四者的相互关系（2 分）？

（2） 请简答系统功能调用和库函数的概念（每个 4 分共 8 分）以及二者的区别（4 分）。

题目 3：（20 分）

请你根据自己的理解回答下列问题：

（1） 处理机调度的多级反馈队列进程调度算法：（a）描述该算法的设计思想（2 分）；（b）总结该算法的必要组成部分（3 分）；（c）分析该算法针对不同类型进程的调度能力、并分别举例说明调度过程（5 分）。

（2） 死锁：（a）描述死锁的概念（2 分）；（b）描述死锁产生的必要条件（2 分）；（c）分析死锁安全状态和安全序列之间的关系（3 分）；（d）分析银行家算法实现的数据结构、算法过程和算法适用条件（3 分）。

题目 4：（20 分）

1965 年 Dijkstra 提出并解决了一个他称之为哲学家进餐的同步问题。请回答什么是哲学家进餐问题（5 分）？并使用类 C 语言写出 Dijkstra 实现哲学家进餐问题的算法（15 分）。

题目 5：（20 分）

请你根据自己的理解回答下列问题：

（1） 简述下列系统功能调用的功能：fork()、exec()、wait()、exit()、getpid()。（5 分，每个 1 分）

（2） 写出 C 语言 main 函数的完整格式（2 分），并回答 main 函数入口参数和系统功能调用 exec()的关系（1 分），回答 main 函数的返回值和系统功能调用 wait ()的关系（1 分），回答 main 函数入口参数与返回值和 shell 语言中内置变量的关系。（1 分）

（3） 请写出一个你上机时使用了上述五个系统功能调用的程序，简述程序的功能，简述你亲自调试该程序时遇到了哪些问题。（5 分）请不要抄袭，你写的程序不可能和别人的是一样的！

（4） 写一个不少于 200 字的课程总结，包括课程体会，以及对课程教学的建议。（5 分）请不要抄袭，你的课程体会不可能和别人的是一样的！

<题目部分结束，总计 5 道题目>

三、《操作系统(1)（08305011）》课程考核报告

题目 1：

在操作系统发展史上，微内核是一个十分重要的操作系统体系结构，总的来说微内核并不是一个成功的体系结构，在主流的操作系统中，没有采用或者抛弃了微内核结构，但是微内核结构仍然吸引学术界的关注，也存在一些基于微内核构建的操作系统。请你根据自己的理解回答下列问题：

(1) 什么是微内核？（2 分）微内核中包括哪些基本功能。（4 分）

微内核

在计算机科学中，微内核架构是一种内核的设计架构，将内核中最基本的功能保留在内核，而将不需要在核心态执行的功能移到用户态执行，从而降低内核的设计复杂性。而移除内核的操作系统代码根据分层的原则被划分为若干服务程序，它们的执行相互独立，交互则都借助于微内核进行通信。微内核结构将操作系统划分为两大部分：微内核和多个服务器。

微内核(Microkernel，μ-kernel)，是一种精心设计、能实现操作系统最基本核心功能的小型内核，由尽可能精简的程序所组成，以实现一个操作系统所需要的最基本功能。同时，微内核操作系统具有以下四项特点：

1) 足够小的内核

在微内核 OS 中，内核是指精心设计的、能实现现代 OS 最基本的核心功能的部分。微内核并非是一个完整的 OS，而只是操作系统中最基本的部分，它通常用于：①实现与硬件紧密相关的处理；②实现一些较基本的功能；③负责客户和服务器之间的通信。它们只是为构建通用 OS 提供一个重要基础，这样就可以确保把操作系统内核做得很小。

2) 基于客户/服务器模式

微内核 OS 都采用客户/服务器模式，将操作系统中最基本的部分放入内核中，而把操作系统的绝大部分功能都放在微内核外面的一组服务器(进程)中实现。客户与服务器之间是借助微内核提供的消息传递机制来实现信息交互的。

3) 应用“机制与策略分离”原理

机制，是指实现某一功能的具体执行机构。而策略，则是在机制的基础上，借助于某些参数和算法来实现该功能的优化，或达到不同的功能目标。在传统的 OS 中，将机制放在 OS 的内核的较低层，把策略放在内核的较高层次中。而在微内核操作系统中，通常将机制放在 OS 的微内核中。

4) 采用面向对象技术

操作系统是一个极其复杂的大型软件系统，可以通过结构设计来分解操作系统的复杂度，还可以基于面向对象技术中的“抽象”和“隐蔽”原则控制系统的复杂性，并确保操作系统的“正确性”、“可靠性”、“易修改性”、“易扩展性”等，并提高操作系统的设计速度。

微内核中的基本功能

微内核结构通常利用“机制与策略分离”的原理来构造 OS 结构，将机制部分以及与硬件紧密相关的部分放入微内核中。由此可知，微内核通常具有如下几方面的功能：

1) 进程(线程)管理

大多数的微内核 OS，对于进程管理功能的实现，都采用“机制与策略分离”的原理。例如，为实现进程调度功能，须在进程管理中设置一个或多个进程优先级队列；能将指定优先级进程从所在队列中取出，并将其投入执行。由于这一部分属于调度功能的机制部分，应将它放入微内核中。

2) 低级存储器管理

通常在微内核中，只配置最基本的低级存储器管理机制，如用于实现将用户空间的逻辑地址变换为内存空间的物理地址的页表机制和地址变换机制，这一部分是依赖于机器的，因此放入微内核。

而实现虚拟存储器管理的策略，则包含应采取何种页面置换算法，采用何种内存分配与回收策略等，应将这部分放在微内核外的存储器管理服务器中去实现。

3) 中断和陷入处理

大多数微内核操作系统都是将与硬件紧密相关的一小部分放入微内核中处理。此时微内核的主要功能，是捕获所发生的中断和陷入事件，并进行相应的前期处理。如进行中断的现场保护，识别中断的类型，然后将有关的事件的信息转化成消息后，把它发给相关的服务器。由服务器根据中断或陷入的类型，调用相应的处理程序来进行后期处理，故中断和陷入处理也应放入微内核。

4) 其他功能

除进程的调度之外，进程的切换、进程(线程)之间的通信，以及多处理机之间的同步等与进程(线程)管理相关的功能，都应放入微内核中。而对用户进程如何分类，以及优先级的确认方式，则属于策略问题，可将它们放入微内核外的进程管理服务器中。

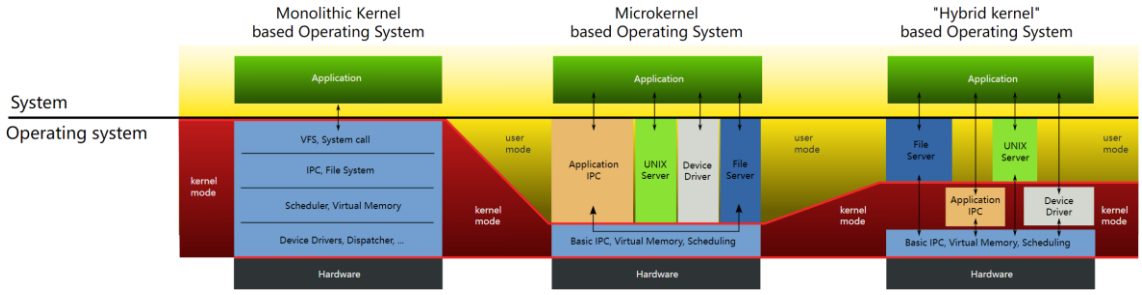

图 1 宏内核、微内核、混合内核的操作系统结构

(2) 和宏内核相比，微内核有哪些优点？（5 分）存在哪些问题？（2 分）

和宏内核相比，微内核的优点

由于微内核操作系统结构是建立在模块化、层次化结构的基础上的，并采用了客户/服务器模式和面向对象的程序设计技术。因此，微内核结构的操作系统具有如下优点：

1) 提高了可扩展性

由于微内核 OS 的许多功能是由相对独立的服务器软件来实现的，当开发了新的硬件和软件时，微内核 OS 只须在相应的服务器中增加新的功能，或再增加一个专门的服务器。与此同时，也必然改善系统的灵活性，不仅可在操作系统中增加新的功能，还可修改原有功能，以及删除已过时的功能，使功能更具有弹性，以形成一个更为精干有效的操作系统。

而与之对比，采用宏内核的操作系统可能需要修改内核程序。

2) 增强了可靠性

这一方面是由于微内核是出于精心设计和严格测试的，容易保证其正确性；另一方面是它提供了规范而精简的应用程序接口(API)，为微内核外部的程序编制高质量的代码创造了条件。此外，由于所有服务器都是运行在用户态，服务器与服务器之间采用的是消息传递通信机制，让服务各自独立，可以减少系统之间的耦合度，易于实现与调试。它可以避免单一组件失效，而造成整个系统崩溃，内核只需要重启这个组件，不至于影响其他服务器的功能，使系统稳定度增加。

同时，就代码数量来看，一般来说，因为功能简化，微核心使用的代码比集成式核心更少，其源代码通常小于 10,000 行。例如，MINIX 3 的源代码少于 6,000 行。更少的代码，也代表更少的潜藏程序 Bug，对于

重视安全性的人来说会较为偏好。

而与之对比，宏内核的各种功能都放在内核中，各个模块共享信息，虽然使性能提高。但是也导致模块之间耦合性提高，某一模块出问题可能会导致整个系统崩溃。

3) 增强了可移植性

随着硬件的快速发展，出现了各种各样的硬件平台，作为一个好的操作系统，必须具备可移植性，使其能较容易地运行在不同的计算机硬件平台上。在微内核结构的操作系统中，所有与特定 CPU 和 I/O 设备硬件有关的代码，均放在内核和内核下面的硬件隐藏层中，而操作系统其它绝大部分(即各种服务器)均与硬件平台无关，因而，把操作系统移植到另一个计算机硬件平台上所需作的修改是比较小的。

而与之对比，宏内核则将大量与硬件有关的内容放入内核中，因此如果要移植到新的平台的话，可能需要较大幅度地内核代码。

4) 提供了对分布式系统的支持

由于在微内核 OS 中，客户和服务器之间以及服务器和服务器之间的通信，是采用消息传递通信机制进行的，致使微内核 OS 能很好地支持分布式系统和网络系统。事实上，只要在分布式系统中赋予所有进程和服务器惟一的标识符，在微内核中再配置一张系统映射表(即进程和服务器的标识符与它们所驻留的机器之间的对应表)，在进行客户与服务器通信时，只需在所发送的消息中标上发送进程和接收进程的标识符，微内核便可利用系统映射表，将消息发往目标，而无论目标是驻留在哪台机器上。

因为所有服务进程都各自在不同地址空间运行，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，要创建一个进程间通信机制，通过消息传递的机制来让服务进程间相互交换消息，调用彼此的服务，以及完成同步。采用主从式架构，使得它在分布式系统中有特别的优势，因为远程系统与本地进程间，可以采用同一套进程间通信机制。

5) 融入了面向对象技术

在设计微内核 OS 时，采用了面向对象的技术，其中的“封装”，“继承”，“对象类”和“多态性”，以及在对象之间采用消息传递机制等，都十分有利于提高系统的“正确性”、“可靠性”、“易修改性”、“易扩展性”等，而且还能显著地减少开发系统所付出的开销。

微内核存在的问题

在微内核 OS 中，由于采用了非常小的内核，以及客户/服务器模式和消息传递机制，使微内核 OS 存在着潜在的缺点。其中最主要的是，较之早期 OS，微内核 OS 的运行效率有所降低。

效率降低的最主要的原因是，在完成一次客户对 OS 提出的服务请求时，需要利用消息实现多次交互和进行用户/内核模式及上下文的多次切换，而宏内核转换的频率会少很多。

在微内核 OS 中，由于客户和服务器及服务器和服务器之间的通信，都需通过微内核，致使同样的服务请求至少需要进行四次上下文切换。第一次是发生在客户发送请求消息给内核，以请求取得某服务器特定的服务时；第二次是发生在由内核把客户的请求消息发往服务器时；第三次是当服务器完成客户请求后，把响应消息发送到内核时；第四次是在内核将响应消息发送给客户时。

实际情况是往往还会引起更多的上下文切换。例如，当某个服务器自身尚无能力完成客户请求，而需要其它服务器的帮助时，其中的文件服务器还需要磁盘服务器的帮助，这时就需要进行八次上下文的切换。

与此同时，进程间通信耗费的资源与时间，比简单的函数调用还多；通常又会涉及到核心空间到用户空间的环境切换(Context Switch)。这使得消息传递有延迟，以及传输量(Throughput)受限的问题，并可能出现性能不佳的问题。

各模块与微内核间通过通信机制交互通信，系统运行效率较低。而在宏内核中模块之间可以直接相互调用。

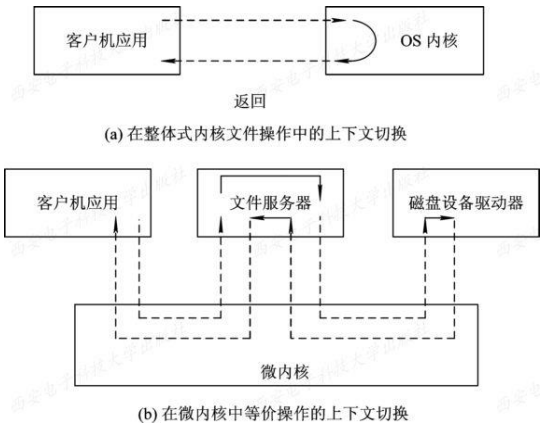

图 2 在传统 OS 和微内核 OS 中的上下文切换

为了改善运行效率，可以重新把一些常用的操作系统基本功能，由服务器移入微内核中。这样可使客户对常用操作系统功能的请求所发生的用户/内核模式和上下文的切换的次数，由四次或八次降为两次。但这又会使微内核的容量明显地增大，在小型接口定义和适应性方面的优点也有所下降，同时也提高了微内核的设计代价。

(3) 请查询资料介绍两个微内核（2 分）和两个宏内核（2 分）的实例。

两个微内核的实例

- **Mach**

微内核结构必然是多线程的，第一代微内核，在内核提供了较多的服务，因此被称为“胖微内核”，它的典型代表是 Mach，它既是 GNU HURD 也是 Mac OS X 的内核。

Mach 是一个由卡内基梅隆大学开发的计算机操作系统微内核，为了用于操作系统之研究，特别是在分布式与并行运算上。是最早实现微核心操作系统的例子之一，是许多其它相似的项目的标准。

Mach 开发项目在卡内基梅隆大学从 1985 年运行到 1994 年，到 Mach 3.0 版结束。其他还有许多人继续 Mach 的研究包括犹他大学的 Mach 4。Mach 的开发是为了取代 BSD 的 UNIX 核心，所以是许多新的操作系统的设计基础。Mach 的研究至今似乎是停止了，虽然有许多商业化操作系统，如 NEXTSTEP 与 OPENSTEP，特别是 Mac OS X(使用 XNU 核心)都是使用 Mach 或其派生系统。Mach 的虚拟内存(VM)系统也被 BSD 的开发者用于 CSRG，并出现在 BSD 派生的系统中，如 FreeBSD。Mac OS X 与 FreeBSD 并未保留 Mach 首倡的微核心结构，除了 Mac OS X 继续提供微核心于内部处理通信以及应用程序直接控制。

- **QNX**

第二代微内核只提供最基本的 OS 服务，典型的 OS 是 QNX，QNX 在黑莓手机 BlackBerry 10 系统中被采用。QNX 是商业类 Unix 实时操作系统，主要针对嵌入式系统市场。该产品开发于 20 世纪 80 年代初，后来改名为 QNX 软件系统公司，公司已被黑莓公司并购。2010 年代后，随着汽车智能化的加速，QNX 在车用市场占有率不断提高，达到 75%

QNX 采取微核心架构，操作系统中的多数功能是以许多小型的 task 来执行，它们被称为 server。这样的架构使得用户和开发者可以关闭不需要的功能，而不需要改变操作系统本身。

QNX Neutrino(2001)已经被移植到许多平台并且运行在嵌入式市场中使用的各种现代处理器上，如 PowerPC 和 x86。QNX 的应用范围极广，包含了：控制保时捷跑车的音乐和媒体功能、福特汽车的 SYNC 3 车载系统、核电站和美国陆军无人驾驶 Crusher 坦克的控制系统，还有 BlackBerry PlayBook 和操作系统。

- **Harmony OS(鸿蒙)**

鸿蒙(HarmonyOS，开发代号 Ark，正式名称为华为终端鸿蒙智能设备操作系统软件)是华为自 2012 年开发的一款可兼容 Android 应用程序的跨平台分布式操作系统。系统性能包括利用“分布式”技术将各款设备融合成一个“超级终端”，便于操作和共享各设备资源。系统架构支持多内核，包括 Linux 内核、LiteOS 和鸿蒙微内核，可按各种智能设备选择所需内核，例如在低功耗设备上使用 LiteOS 内核。

- **其他微内核操作系统**

除此之外，还有以下操作系统采用了微内核结构：

- FreeRTOS：微内核，实时操作系统(RTOS)
- Redox：Rust 实现的微内核操作系统
- Minix：迷你版本的类 Unix 操作系统，采用微核心设计
- INTEGRITY：微内核实时操作系统(RTOS)
- L4 微内核系列：L4 系列，小而快的开源微内核

两个宏内核的实例

1) Linux

Linux 是一种自由和开放源码的类 UNIX 操作系统。该操作系统的内核由林纳斯·托瓦兹在 1991 年 10 月 5 日首次发布，再加上用户空间的应用程序之后，就成为了 Linux 操作系统。基于 Linux 的系统是一个模块化的类 Unix 操作系统。Linux 操作系统的大部分设计思想来源于 20 世纪 70 年代到 80 年代的 Unix 操作系统所创建的基本设计思想。

Linux 系统使用宏内核，由 Linux 内核负责处理进程控制、网络，以及外围设备和文件系统的访问。在系统运行的时候，设备驱动程序要么与内核直接集成，要么以加载模块形式添加。Linux 具有设备独立性，它内核具有高度适应能力，从而给系统提供了更高级的功能。

2) Unix

Unix 操作系统，是一个强大的多用户、多任务操作系统，支持多种处理器架构，按照操作系统的分类，属于分时操作系统，最早由肯·汤普逊、丹尼斯·里奇和道格拉斯·麦克罗伊于 1969 年在 AT&T 的贝尔实验室开发。Unix 因为其安全可靠，高效强大的特点在服务器领域得到了广泛的应用。直到 GNU/Linux 流行开始前，Unix 也是科学计算、大型机、超级计算机等所用操作系统的主流。现在其仍然被应用于一些对稳定性要求极高的数据中心之上。

(4) 如果由你设计实现一个操作系统内核，你是采用微内核还是宏内核？给出你的理由。（3 分）

如果由我设计实现一个操作系统内核，我会选择**采用微内核**。

随着智能手机，移动互联网终端 MID、物联网 IoT 等一系列概念的推出与发展，嵌入式系统正逐渐超越个人计算机成为主流的个人信息交互终端。虽然目前大部分主流移动平台的操作系统都采用宏内核或混合内核架构的操作系统，如 Android 和 iOS，但在物联网 IoT 设备中，由于其设备的复杂多样性远超前者，选择微内核会更具有优势。微内核将所有与特定 CPU 和 I/O 设备硬件有关的代码，均放在内核和内核下面的硬件隐藏层中，而操作系统其它绝大部分(即各种服务器)均与硬件平台无关，因而，把操作系统移植到另一个计算机硬件平台上所需作的修改是比较小的，即相比宏内核，可以跟轻松地移植到新的硬件平台上。例如在本学期中，我选修了《单片机技术》这门课程，其中所使用的嵌入式开发平台的系统，正是采用了基于为内核的 μC/OS-II，在这门课程的实验过程中，我也体验到了微内核操作系统的优美和精巧。

与此同时，拥有微内核架构的操作系统被设计用以构成一个可自由裁减的系统并且易于保证其可信计算基础，而且系统所定义的操作系统和用户态应用程序之间的接口，与硬件平台的接口很相近，这样既保证了操作系统的功能性，又避免了操作系统的代码过于冗长，十分适合用以构建一个良好的虚拟机环境。

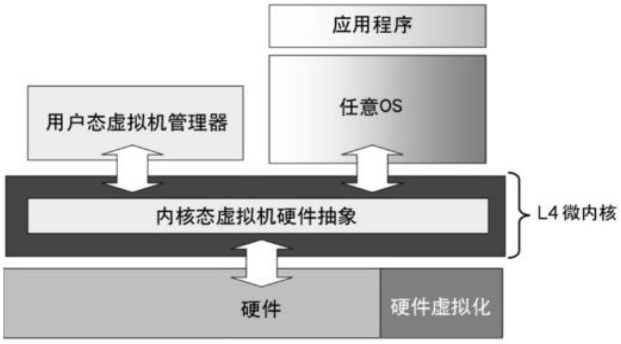

图 3 基于微内核架构的虚拟机的系统结构

而嵌入式系统也面临信息安全性能所提出的挑战。通过虚拟化，我们可以支持多个相互隔离的操作系统在嵌入式平台上的同时运行，把如电子商务等安全性要求较高的应用封装在一个高度隔离的客户操作系统当中，并开放另一个操作系统作为普通应用程序的运行环境，实现高度安全可靠的运行模式。

而通过使用基于微内核架构的虚拟机，我们可以把嵌入式系统的硬件资源转化为各个不同的实时系统服务向上以虚拟设备的方式提供给虚拟机上运行的客户操作系统。借由这种方式，可以同时支持丰富的非实时和实时应用程序的同时运行并为非实时的应用程序与实时的系统功能之间提供一个良好而透明的交互接口。

因此，如果由我来设计一个操作系统内核，我会选择微内核。

题目 2：

请你根据自己的理解回答下列问题：

(1) 请简答核心态、用户态、特权指令、普通指令四个概念（每个 2 分共 8 分），以及四者的相互关系（2 分）？

核心态、用户态、特权指令、普通指令

1) 核心态

又称为管态、内核态或系统态，它具有较高的特权，能执行全部指令(包括特权指令)，访问所有的寄存器、存储器以及外部设备，传统的 OS 较多在核心态运行。

2) 用户态

又称为目态，它具有较低特权的执行状态，仅能执行规定的指令，访问指定的寄存器和存储区。因此用户态提供给用户程序使用，防止用户程序对进行危险操作，给操作系统带来安全隐患。

3) 特权指令

在 CPU 设计和生产的时候，就划分了特权指令和非特权指令。特权指令是在核心态运行的指令，它对内存空间的访问范围基本不受限制，不仅能访问用户空间，还能访问系统空间, 一般只会授权操作系统或者其他系统软件使用。

它主要用于系统资源的分配和管理，包括改变系统工作方式，检测用户的访问权限，修改虚拟存储器管理的段表、页表，完成任务的创建和切换等。常见的特权指令有以下几种：①有关对 I/O 设备使用的指令，如启动 I/O 设备指令、测试 I/O 设备工作状态和控制 I/O 设备动作的指令等；②有关访问程序状态的指令，如对程序状态字(PSW)的指令等；③存取特殊寄存器指令，如存取中断寄存器、时钟寄存器等指令。④其他。

4) 普通指令

在用户态运行的指令。应用程序所使用的都是非特权指令，它只能完成一般性的操作和任务，不能对系统中的硬件和软件直接进行访问，对内存的访问范围也局限于用户空间。当然操作系统的内核自然也能使用这些普通指令。

四者的相互关系

当 CPU 处于核心态时，CPU 可以执行全部的指令，包括特权指令和用户指令。

当 CPU 处于用户态时，CPU 只能执行用户指令。

CPU 中有一个寄存器——程序状态字寄存器(PSWR)。可人为规定：当该寄存器的值是 0 时，表示 CPU 处于用户态，而当为 1 时，表示 CPU 处于核心态(或相反规定)。CPU 正是通过程序状态字寄存器，知道自己所处的状态。

CPU 从内核态切换为用户态：执行一条特权指令，即修改 PSW 的标志位为“用户态”对应的值，从而切换至用户态，此时操作系统将主动让出 CPU 使用权。

CPU 从用户态切换为内核态：由“中断”引发(包含外中断和内中断)，硬件自动完成状态转变的过程，触发中断信号意味着操作系统将强行夺回 CPU 的使用权，需要操作系统介入的地方，都会触发中断信号。如：非法使特权指令，应用程序申请操作系统提供服务等。

下图直接地展示了四者的相互关系：

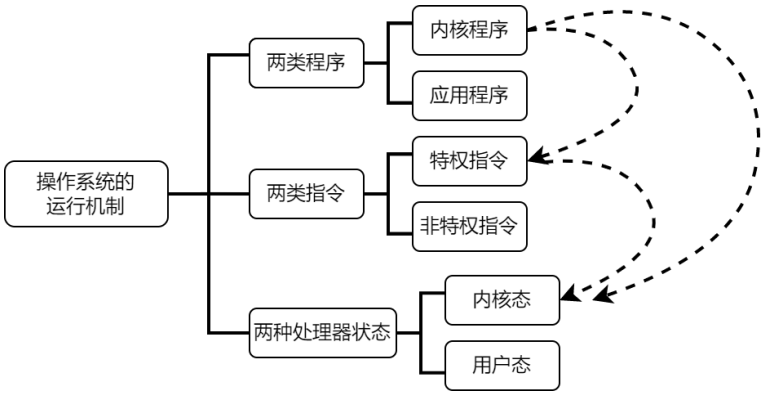

图 4 两类程序、两类指令、两种处理器状态的相互关系

(2) 请简答系统功能调用和库函数的概念（每个 4 分共 8 分）以及二者的区别（4 分）。

系统功能调用

系统调用是操作系统提供给应用程序(程序员/编程人员)使用的接口，可以理解为一种可供应用程序调用的特殊函数，应用程序可以发出系统调用请求来获得操作系统的服务。

操作系统作为用户和计算机硬件之间的接口，需要向上提供一些简单易用的服务。主要包括命令接口和程序接口。其中，程序接口由一组系统调用组成。

应用程序通过系统调用请求操作系统的服务。系统中的各种共享资源都由操作系统统一掌管，因此在用户程序中，凡是与资源有关的操作(如存储分配、I/O 操作、文件管理等)，都必须通过系统调用的方式向操作系统提出服务请求，由操作系统代为完成。这样可以保证系统的稳定性和安全性，防止用户进行非法操作。

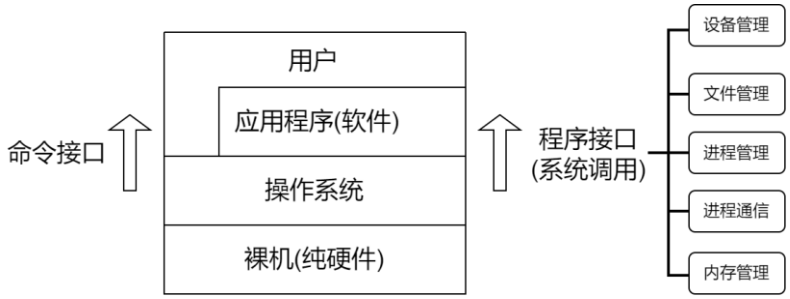

图 5 系统调用

系统调用的相关处理涉及到对系统资源的管理、对进程的控制、这些功能需要执行一些特权指令才能完成，因此系统调用的相关处理需要在核心态下进行。

库函数

库函数是编程语言给应用程序(程序员/编程人员)提供的接口，有些库函数将系统调用封装起来，隐藏系统调用的一些细节，使程序员编程更加方便，如“创建一个新文件”的函数涉及系统调用。当然，也有不使用系统调用的库函数，如：求绝对值。

二者的区别

常见系统调用和库函数：

- 常见系统调用

  open, close, read, write, ioctl、fork、clone、exit、getpid、access、chdir、chmod、stat、brk、mmap 等，需要包含 unistd.h 等头文件。

- 常见库函数

  printf、scanf、fopen、fclose、fgetc、fgets、fprintf、fsacnf、fputc、calloc、free、malloc、realloc、strcat、strchr、strcmp、strcpy、strlen、strstr 等，需要包含 stdio.h、string.h、alloc.h、stdlib.h 等头文件。

二者的区别在于：

1) 普通应用程序既可以直接进行系统功能调用，也可以使用库函数进行系统功能调用

2) 库函数中并不都涉及系统功能调用

不涉及系统调用的库函数：如的“取绝对值”的函数。而涉及系统调用的库函数：如“创建一个新文件”的函数。

3) 库函数是编程语言或者应用程序的一部分，而系统调用是操作系统的一部分

4) 系统调用通常不可替换，而库函数通常可替换

普通的库函数调用由函数库或用户自己提供，因此库函数是可以替换的。例如，对于存储空间分配函数 malloc，如果不习惯它的操作方式，我们完全可以定义自己的 malloc 函数。

5) 系统调用通常提供最小接口，而库函数通常提供较复杂功能

例如 brk、sbrk 系统调用分配一块空间给进程，而 malloc 则在用户层次对这以空间进行管理。

6) 系统调用运行在内核空间，而库函数运行在用户空间

因为系统调用属于内核，和库函数不属于内核。因此，如果当用户态进程调用一个系统调用时，CPU 需要将其切换到内核态，并执行一个内核函数。

7) 内核调用都返回一个整数值，而库函数并非一定如此

在内核中，整数或 0 表示系统调用成功结束，而负数表示一个出错条件。而出错时，内核不会将其设置为 errno，而是由库函数从系统调用返回后对其进行设置或使用。

8) POSIX 标准针对库函数而不是系统调用

判断一个系统是否与 POSIX 需要看它是否提供一组合适的应用程序接口，而不管其对应的函数是如何实现的。因此从移值性来讲，使用库函数的移植性较系统调用更好。

系统调用运行时间属于系统时间，库函数运行时间属于用户时间

9) 调用系统调用开销相对库函数来说更大

很多库函数本身都调用了系统调用，但直接调用系统调用的开销较大。这是得益于双缓冲的实现，在用户态和内核态，都应用了缓冲技术，对于文件读写来说，调用库函数可以大大减少调用系统调用的次数。而用户进程调用系统调用需要在用户空间和内核空间进行上下文切换，开销较大。因此，库函数的开销也就会比直接调用系统调用小了。另外一方面，库函数同样会对系统调用的性能进行优化。

系统调用与库函数有联系也有区别，但是通常情况下，会建议使用库函数，主要出于双缓冲技术、移植性以及系统调用本身性能缺陷的考虑。

题目 3：

请你根据自己的理解回答下列问题：

(1) 处理机调度的多级反馈队列进程调度算法：(a)描述该算法的设计思想（2 分）；(b)总结该算法的必要组成部分（3 分）；(c)分析该算法针对不同类型进程的调度能力、并分别举例说明调度过程（5 分）。

多级反馈队列进程调度算法的设计思想

多级反馈队列调度算法不必事先知道各种进程所需的执行时间，还能较好地满足各种类型进程的需要，是目前公认的一种较好的进程调度算法。

多级反馈队列进程调度算法的设计思想如下：

- 设置多个就绪队列，并为各个队列赋予不同的优先级和不同长度的时间片。
- 新创建的进程挂到第一优先级的队列后，然后按 FCFS 原则排队等待调度。当轮到其执行时，如它能在时间片内完成，便撤离系统；如果不能完成，便被挂入第二级队列后，……，最后一级队列采用时间片轮转法。
- 仅当第一级队列空闲时，调度程序才调度第二级队列中的进程运行，依次类推……；新进程可抢占低级进程的处理机。
- 其算法思路的核心是：既不让需要长时间完成的进程一直被冷落，也不让短时间的进程长时间等待。

多级反馈队列进程调度算法的详细设计思想如下所述：

1) 设置多个就绪队列

设置多个就绪队列。在系统中设置多个就绪队列，并为每个队列赋予不同的优先。第一个队列的优先级最高，第二个次之，其余队列的优先级逐个降低。该算法为不同列中的进程所赋予的执行时间片的大小也各不相同，在优先级愈高的队列中，其时间片愈小。

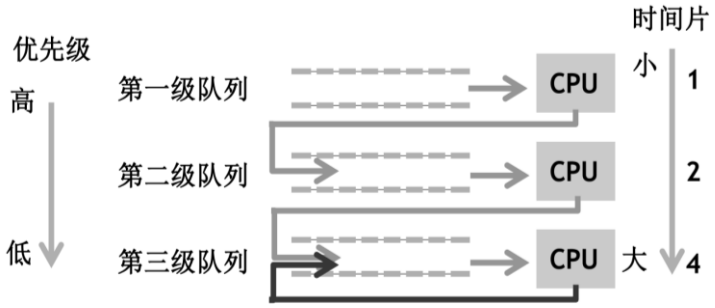

图 6 多级反馈队列调度算法

2) 每个队列都采用 FCFS 算法

当新进程进入内存后，首先将它放入第一队列的末尾，按 FCFS 原则等待调度。当轮到该进程执行时，如它能在该时间片内完成，便可撤离系统。否则，即它在一个时间片结束时尚未完成，调度程序将其转入第二队列的末尾等待调度；如果它在第二队列中运行个时间片后仍未完成，再依次将它放入第三队列……依此类推。当进程最后被降到第 n 队列后，在第 n 队列中便采取按 RR(时间片轮转)方式运行。

3) 按队列优先级调度

调度程序首先调度最高优先级队列中的诸进程运行，仅当第一队列空闲时才调度第二队列中的进程运行；换言之，仅当第 1~(i-1)所有队列均空时，才会调度第 i 队列中的进程运行。如果处理机正在第 i 队列中为某进程服务时又有新进程进入任一优先级较高的队列，此时须立即把正在运行的进程放回到第 i 队列的末尾，而把处理机分配给新到的高优先级进程。

多级反馈队列进程调度算法的必要组成部分

1) 进程控制符(PID)

2) 进程名字(Name)

3) 进程到达时间(Time)

4) 进程执行时间(UseTime)

5) 就绪队列

设置多个就绪队列，为存放进程的数据结构，采用先入先出 FIFO。

6) 优先级标识

通过为每个队列赋予不同的优先，控制 CPU 优先取出哪个队列的进程进行工作

7) 时间片长度

不同优先级的时间片长度必须合理分配，倘若每个时间片长度一样，则多级反馈队列进程调度算法则会退化成时间片轮转 RR 算法。

分析该算法针对不同类型进程的调度能力、并分别举例说明调度过程

多级反馈队列调度算法能较好地满足各种类型用户(进程)的需要。

1) 终端型用户

由于终端型用户的进程一般很小，在该算法中一般在第一优先级队列中被运行后就可以直接完成，不需要再下放到第二优先级队列，所以对于终端型用户的进程该算法能及时完成。

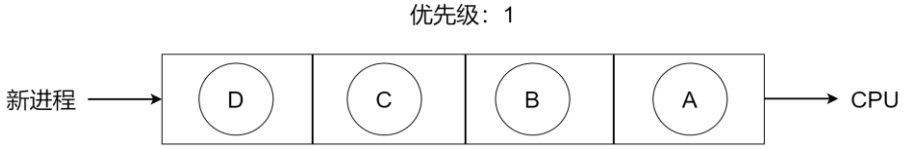

图 7 终端型用户

2) 短批处理作业用户

对于短批处理作业用户而言，如果在第一优先队列中被运行后没有结束，那么正常下放到第二优先级队列，第三或者更低优先级的队列就能完成，仍然保持了较短的周转周期。

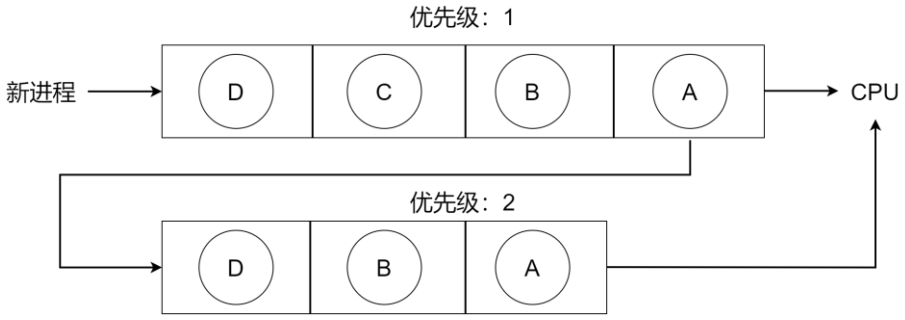

图 8 短批处理作业用户

3) 长批处理作业用户

长批处理作业用户的进程一般很大，因此很可能会被下放到优先级为 n 的队列中，此时队列中采用 RR 法进行处理进程，对长批处理作业用户而言，该算法保证了其进程不会被长期冷落。

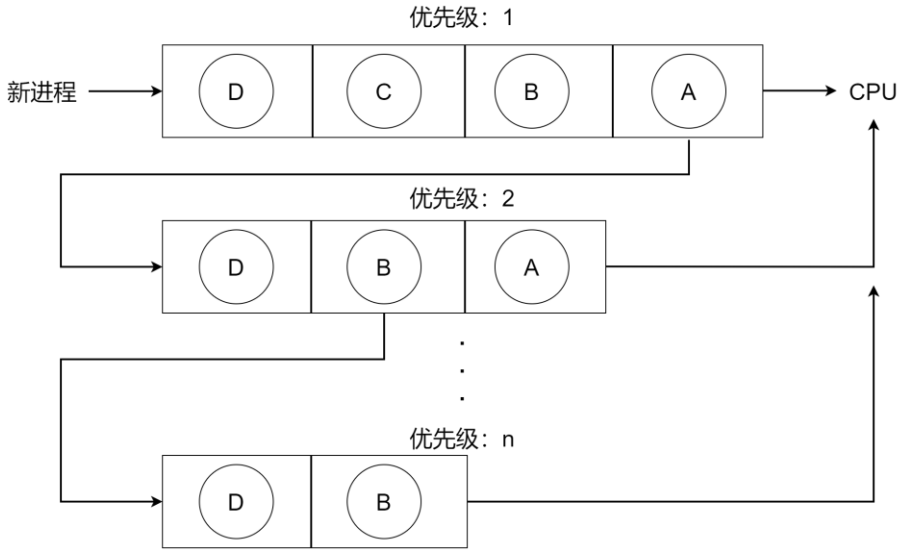

图 9 长批处理作业用户

如图所示，B 和 D 是长批处理作业的进程，其由于始终没有运行完毕，会被不断下放到优先级为 n 的队列中。其中 B 和 D 会执行 RR 直到进程运行完毕为止。

选用以下这个较为具体的例子来说明该算法针对不同类型进程的调度能力。

1) 长进程与短进程混合类型

先设置三个多级队列，如下表所示：

队列 1	队列 2	队列 3
2	4	8

初始化进程，如下表所示：

进程名称	到达时间	调度时间
进程 1	0	20
进程 2	2	4
进程 3	3	5

共有一个长进程和两个短进程。

运行结果如下表所示：

进程名称	到达时间	服务时间	完成时间	周转时间	带权周转时间
进程 1	0	20	29	29	1.45
进程 2	2	4	12	10	2.5
进程 3	3	5	15	12	2.4

平均周转时间为 17，平均带权周转时间为 2.12。

可见，在多级反馈队列调度算法下，每个新到达的进程都可以很快就得到响应，进程 2 和 3 都在到达时或下一时刻就被调度；短进程 2 和 3 只用较少的时间就可完成，分别是 12 时间和 15 个时间；长进程 1，虽然一开始就被调度但是由于多级反馈队列调度算法的特性，无法一直占据 CPU，避免其他进程饥饿。

2) 全短进程类型

先设置三个多级队列，如下表所示：

队列 1	队列 2	队列 3
2	4	8

初始化进程，如下表所示：

进程名称	到达时间	调度时间
进程 1	0	4
进程 2	2	1
进程 3	3	2

共有一个长进程和两个短进程。

运行结果如下表所示：

进程名称	到达时间	服务时间	完成时间	周转时间	带权周转时间
进程 1	0	4	7	7	1.75
进程 2	2	1	3	1	1
进程 3	3	2	5	2	1

平均周转时间为 3.33，平均带权周转时间为 1.25。

可见，在多级反馈队列调度算法下，短进程只用较少的时间就可完成。

(2) 死锁：(a)描述死锁的概念（2 分）；(b)描述死锁产生的必要条件（2 分）；(c)分析死锁安全状态和安全序列之间的关系（3 分）；(d)分析银行家算法实现的数据结构、算法过程和算法适用条件（3 分）。

死锁的概念

如果一组进程(两个或以上)中的每一个进程都在等待仅由该组进程中的其它进程才能引发的事件，那么该组进程是死锁的(Deadlock)。

在一组进程发生死锁的情况下，这组死锁进程中的每一个进程，都在等待另一个死锁进程所占有的资源。或者说每个进程所等待的事件是该组中其它进程释放所占有的资源。但由于所有这些进程已都无法运行，因此它们谁也不能释放资源，致使没有任何一个进程可被唤醒。这样这组进程只能无限期地等待下去。

死锁产生的必要条件

产生死锁必须同时具备下面四个必要条件，只要其中任一个条件不成立，死锁就不会发生：

1) 互斥条件

进程对所分配到的资源进行排他性使用，即一段时间内，某资源只能被一个进程占用。如果此时还有其他进程请求该资源，则进程只能等待，直至占有资源的进程用毕释放。

2) 请求和保持条件

进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源已经被其他进程占有，此请求进程被阻塞，但对自己已获得的资源保持不放。

3) 不可抢占条件

进程已获得的资源在未使用完成之前不能被抢占，只能在进程使用完成时由自己释放。

4) 循环等待条件

发生死锁时，必存在一个进程——资源的循环链，即进程的集合{P_0，P_1，P_2，…P_n}中的 P_0 正在等待 P_1 占用的资源，P_1 正在等待 P_2 占用的资源，……P_n 正在等待 P_0 占用的资源。**值得注意的是，发生死锁时一定有循环等待，但是发生循环等待时未必死锁。**

死锁安全状态和安全序列之间的关系

所谓安全状态，是指系统能够按某种进程推进顺序(P_1,P_2,…,P_n)为每个进程分配其所需资源，直至满足每个进程对资源的最大需求，使每个进程都可顺利的执行完成。其中进程的推进顺序(P_1,P_2,…,P_n)被称为安全序列。如果系统中能找到这样一个安全序列，则称系统处于安全状态。

因此，当我们需要为某个进程分配资源时，应该考虑是否分配后是否还存在一个安全序列，如果有则可以放心分配。如果系统中无法找到一个安全序列，则称系统处于不安全状态，不能分配。

下面通过一个例子，来说明安全状态与安全序列的关系。假定系统中有三个进程 P_1、P_2 和 P_3，共有 12 台磁带机。进程 P_1 总共要求 10 台磁带机，P_2 和 P_3 分别要求 4 台和 9 台。假设在 T_0 时刻，进程 P_1、P_2 和 P_3 已分别获得 5 台、2 台和 2 台磁带机，尚有 3 台空闲未分配，如下表所示：

进程	最大需求	已分配	可用
P_1	10	5	3
P_2	4	2	
P_3	9	2	

对于此情况，可以知道 T_0 时刻系统是安全的，因为存在一个安全序列{P_2,P_1,P_3}。

如果不按照安全序列分配资源，则系统可能会有安全状态进入不安全状态。例如，在 T_0 时刻后 T_1 时刻，P_3 又请求一台磁带机，如果此时分配资源给进程 P_3，此时的资源分配如下表所示：

进程	最大需求	已分配	可用
P_1	10	5	2
P_2	4	2	
P_3	9	3	

可以发现，在 T_1 时刻，无法找到一个安全序列，因此在 T_1 时刻系统处于不安全状态。

银行家算法实现的数据结构、算法过程和算法适用条件

数据结构

设系统中有 *m* 类资源，*n* 个进程

1) 可利用资源向量 *Available*

可利用资源向量 *Available* 含有 *m* 个元素的一维数组，每个元素代表一类可利用的资源数目。如果 $Available_{[j]}=k$,表示系统中现有 R_j 类资源 *k* 个。

2) 最大需求矩阵 *Max*

最大需求矩阵 *Max* 是一个 *n*×*m* 的矩阵，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。如果 $Max_{[i,j]}=k$，则表示进程 *i*，需要 R_j 类资源的最大数目为 *k*。

3)* 分配矩阵 *Allocation

分配矩阵 Allocation 为 *n*×*m* 的矩阵，它定义了系统中每类资源当前已分配给每个进程的资源数。$Allocation_{[i,j]}=K$,表示进程 i 当前已分得 R_j 类资源的数目为 *K*。

4) 需求矩阵 Need

需求矩阵 *Need* 为 *n*×*m* 的矩阵，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 R_j 类资源 *K* 个，方能完成其任务。

算法过程

设 $Requesti_{[j]} = k$，表示进程 P_i 需要 *k* 个 R_j 类型的资源

1) 如果 $Requesti_{[j]} <= Need_{[i,j]}$，便转向步骤 2；否则认为出错，因为它所请求的资源数已超过它所需要的最大值。

2) 如果 $Requesti_{[j]} <= Available_{[j]}$，便转向步骤 3；否则，表示尚无足够资源，$P_i$ 需等待。

3) 系统试探着把资源分配给进程 Pi ，并修改下面数据结构中数值：

$$Available_{[j]} = Available_{[j]} - Requesti_{[j]};$$

$$Allocation_{[i,j]} = Allocation_{[i,j]} + Requesti_{[j]};$$

$$Need_{[i,j]} = Need_{[i,j]} - Requesti_{[j]};$$

4) 系统执行安全性算法，检查此次资源分配后系统是否处于安全状态以决定是否完成本次分配。

设置两个向量：

工作向量 *work*：表示系统可提供给进程继续运行所需的各类资源数目，含有 *m* 个元素的一维数组，初始时，*work = Available*;

Finish：含 *n* 个元素的一维数组，表示系统是否有足够的资源分配给 *n* 个进程，使之运行完成。开始时先令 $Finish_{[i]} = false$ (*i=1..n*); 当有足够源分配给进程 i 时，再令 $Finish_{[i]} = true$。

5) 从进程集合中找到一个能满足下述条件的进程：

$Finish_{[i]} = false$; $Need_{[i,j]} <= work_{[j]}$; 若找到，执行步骤 3，否则执行步骤 4。

6) 当进程 P_i 获得资源后，可顺利执行，直至完成，并释放出分配给它的资源，故应执行：

$$work_{[j]} = work_{[j]} + Allocation_{[i,j]};$$

$$Finish_{[i]} = true;$$

goto step (*2*);

7) 如果所有进程的 $Finish_{[i]} = true$ 都满足，则表示系统处于安全状态；否则，系统处于不安全状态

算法适用条件

如果要使用银行家算法，那么每一个新的进程进入系统时，它必须申明在运行过程中，可能需要每种资源类型的最大单元数目，其数目不应超过系统所拥有的资源总量。

题目 4：

1965 年 Dijkstra 提出并解决了一个他称之为哲学家进餐的同步问题。请回答什么是哲学家进餐问题（5 分）？并使用类 C 语言写出 Dijkstra 实现哲学家进餐问题的算法（15 分）。

哲学家进餐问题

哲学家进餐问题描述有五个哲学家共用一张圆桌，分别坐在周围的五张椅子上，在圆桌上有五个碗和五只筷子，他们的生活方式是交替地进行思考和进餐。平时，一个哲学家进行思考，饥饿时便试图取用其左右最靠近他的筷子，只有在他拿到两只筷子时才能进餐。进餐毕，放下筷子继续思考。

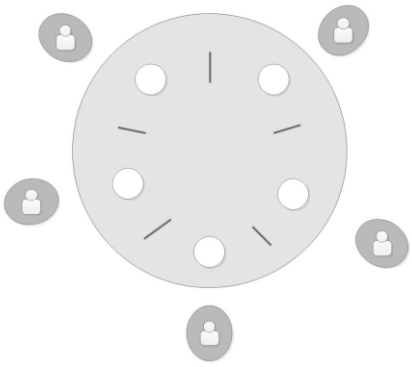

图 10 哲学家进餐问题示意图

哲学家进餐问题的表层含义，正是以上所述的内容。而其背后的深层含义，则是一个需要避免死锁的问题，假设每个哲学家都不管不顾，同时拿起左边的筷子，就会陷入死锁状态，因此需要算法来避免此局面。

更进一步地，哲学家进餐问题可以抽象为：假定系统中有 5 个进程{P_0, P_1, P_2, P_3, P_4}和 5 种资源{C_0, C_1, C_2, C_3, C_4}。则对于进程 P_i，其必须在获取了资源 C_i 和 $C_{(i+1)\%5}$ 后才能运行。此时哲学家进餐问题就转变为了如何合理分配资源，避免发生死锁的问题。哲学家问题可以归纳为在工程中需要考虑避免的死锁情况。

Dijkstra 实现哲学家进餐问题的算法(类 C 语言)

1) 利用记录型信号量

```
semaphore chopstick[5] = {1, 1, 1, 1, 1}; //定义信号量数组 chopstick[5]，并初始化

Pi() { //i 号哲学家的进程
    while (true) {
        P(chopstick[i]); //取左边的筷子
        P(chopstick[(i + 1) % 5]); //取右边的筷子
        Eat; //进餐
        V(chopstick[i]); //放回左边的筷子
        V(chopstick[(i + 1) % 5]); //放回右边的筷子
        Think; //思考
    }
}
```

此算法虽能保证相邻两位不会同时进餐，但有可能引起死锁。该算法存在以下问题：当 5 名哲学家都想要进餐并分别拿起左边的筷子时(都恰好执行完 P(chopstick[i]);)，筷子已被拿完，等到他们再想拿做右边的筷子时(执行 P(chopstick[(i+1)%5]);)就全被阻塞，因此此时出现了死锁。

2) 最多允许四个哲学家同时进餐

```
semaphore chopstick[5] = {1, 1, 1, 1, 1}; //定义信号量数组 chopstick[5]，并初始化
semaphore count = 4; //资源信号量，最多允许 4 人同时取筷子

Pi() { //i 号哲学家的进程
    while (true) {
        P(count);
        P(chopstick[i]); //取左边的筷子
        P(chopstick[(i + 1) % 5]); //取右边的筷子
        Eat; //进餐
        V(chopstick[i]); //放回左边的筷子
        V(chopstick[(i + 1) % 5]); //放回右边的筷子
        Think; //思考
    }
}
```

为了防止死锁发生，可对哲学家进程施加一些限制条件。此算法则是限制最多允许四个哲学家同时进餐。这样可以保证至少有一个哲学家是可以拿到左右两只筷子的。

3) 仅当一名哲学家的左右筷子可用时才允许抓取

```
semaphore chopstick[5] = {1, 1, 1, 1, 1}; //定义信号量数组 chopstick[5]，并初始化
semaphore mutex = 1; //设置取筷子的互斥信号量，互斥地取筷子

Pi() { //i 号哲学家的进程
    while (true) {
        P(mutex); //在取筷子前获得互斥量
        P(chopstick[i]); //取左边的筷子
        P(chopstick[(i + 1) % 5]); //取右边的筷子
        V(mutex); //释放取筷子的信号量
        Eat; //进餐
        V(chopstick[i]); //放回左边的筷子
        V(chopstick[(i + 1) % 5]); //放回右边的筷子
        Think; //思考
    }
}
```

为了防止死锁发生，可对哲学家进程施加一些限制条件。此算法则是限制仅当一名哲学家左右两边的筷子都可用时，才允许他抓起筷子。

4) 对哲学家顺序编号

```
semaphore chopstick[5] = {1, 1, 1, 1, 1}; //定义信号量数组 chopstick[5]，并初始化

Pi() { //i 号哲学家的进程
    while (true) {
        if (i % 2 == 1) {
            P(chopstick[i]); //取左边的筷子
            P(chopstick[(i + 1) % 5]); //取右边的筷子
        } else {
            P(chopstick[(i + 1) % 5]); //取右边的筷子
            P(chopstick[i]); //取左边的筷子
        }
        Eat; //进餐
        V(chopstick[i]); //放回左边的筷子
        V(chopstick[(i + 1) % 5]); //放回右边的筷子
        Think; //思考
    }
}
```

为了防止死锁发生，可对哲学家进程施加一些限制条件。此算法对哲学家顺序编号，要求奇数号哲学家先拿左边的筷子，然后拿右边的筷子，而偶数号哲学家刚好相反。此做法的目的是破坏循环等待条件，奇偶分开申请筷子顺序，避免死锁。

5) 利用 AND 信号量机制解决哲学家进餐问题

```
semaphore chopstick[5] = {1, 1, 1, 1, 1}; //定义信号量数组 chopstick[5]，并初始化

Pi() { //i 号哲学家的进程
    while (1) {
        Swait(chopstick[(i + 1) % 5], chopstick[i]); //取左边的筷子
        Eat; //进餐
        Ssignal(chopstick[(i + 1) % 5], chopstick[i]); //放回右边的筷子
        Think; //思考
    }
}
```

在哲学家进餐问题中，要求每个哲学家先获得两个临界资源(筷子)后方能进餐。本质上是 AND 同步问题。此算法的核心思想是破坏请求和保持条件，一次性申请完这个进程在整个运行过程中所需的全部资源。

题目 5：

请你根据自己的理解回答下列问题：

(1) 简述下列系统功能调用的功能：fork()、exec()、wait()、exit()、getpid()。（5 分，每个 1 分）

表 1 五个系统调用

系统调用	需要的头文件	功能	特点
fork()	#include<unistd.h>	fork 系统调用用于创建子进程，它与父进程(系统调用 fork 的进程)同时运行。	子进程创建后，内存给子进程创建 task_struct，父进程在创建子进程时遵守的规则是：“读时共享，写时复制”。代码是共享的(fork 之后的代码，父子进程共享)，数据是各自私有一份。
exec()	#include<unistd.h>	进程调用 exec 函数族可以执行新的程序。	exec 函数族使得另一个可执行程序的数据段、代码段和堆栈段取代当前进程的。但并不创建新进程，PID 不改变。而且调用 exec 函数，新程序执行结束后就结束，不会执行原子进程后面的代码。
wait()	#include <sys/wait.h> #include <sys/types.h>	wait 系统调用能阻塞父进程，并(在子进程结束后)回收子进程。	调用 wait 函数父进程。 ● 阻塞等待子进程退出。 ● 回收子进程残留资源。父进程回收，回收的是 PCB（储存子进程结束原因等信息）。 ● 获取子进程结束状态(退出原因)。
exit()	#include <stdlib.h>	exit 系统调用会关闭进程中的所有文件，终止调用 exit 函数的进程	在 Linux 中，exit()的返回参数会被传递操作系统，以供其他程序使用
getpid()	#include<unistd.h> #include <sys/types.h>	获取调用 getpid 的进程 PID	无

(2) 写出 C 语言 main 函数的完整格式（2 分），并回答 main 函数入口参数和系统功能调用 exec()的关系（1 分），回答 main 函数的返回值和系统功能调用 wait ()的关系（1 分），回答 main 函数入口参数与返回值和 shell 语言中内置变量的关系。（1 分）

C 语言 main 函数的完整格式

当初学 C 语言时，我曾觉得 main 作为整个程序的入口函数，是不需要传递参数的，但事实上，经过查阅资料后我发现，完全可以给 main()传入参数进而控制整个程序的执行，就像使用 DOS 命令传入的参数一样，这里面 argc 表示传入的参数的个数，包括命令本身。argv 是一个字符串数组，即每一个元素都是一个字符串(的首地址)，命令本身是 argv[0]，依次类推。

```
//main 函数的完整形式
int main(int argc, const char *argv[]) {
    //函数体
    return 0;
}
```

值得注意的是，上面所述的含有两个参数的 main 函数并不是完整的 main 函数，完整的 main 函数有第三个参数 char *envp[]

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

第三个参数用来在程序运行时获取系统的环境变量，操作系统运行程序时通过 envp 参数将系统环境变量传递给程序。

因此，总结如下：

- C 语言 main 函数的完整格式为 int main(int argc,char * argv[],char * envp[])
- argc：是 argument coun 的缩写，表示的是传入参数的个数
- char* argv[]：是 argument vector 的缩写，保存运行时传递 main 函数的参数，argv 是一个字符串数组，其数组元素的类型是字符指针，每一个字符指针都指向一个特定的字符串(指向一个命令行参数)
- char* envp[]：也是一个字符串数组，主要保存用户环境中的变量字符串，以 NULL 结束

main 函数入口参数和系统功能调用 exec()的关系

操作系统内核执行 C 程序时会使用一个 exec 函数，启动一个特殊的启动例程(StartupRoutine)获取 main 函数的命令行参数 argc 和 argv，为 main 函数的执行做准备。

此处先给出结论：main 函数入口参数和 exec 函数关系为 main 函数第二个参数为 exec 函数第一个参数和第二个参数。

- 第一个参数 argc 不需要传递，argc 是 argv 的大小，程序将自动计算。
- 第二个参数 char *argv[]是指向字符串的指针数组。
  - argv[0]：指向程序运行的全路径名
  - argv[1]：指向执行程序名后的第一个字符串，表示真正传入的第一个参
  - argv[2]：指向执行程序名后的第二个字符串，表示传入的第二个参数
  - 规定：argv[argc]为 NULL，表示参数的结尾

以下对上述内容进行详细分析。

main 函数的第二个参数和 exec 函数族的第二个参数有极大的相似和关联性，以下对此内容进行分析。

exec 函数族就提供了一个在进程中启动另一个程序执行的方法。它可以根据指定的文件名或目录名找到可执行文件，并用它来取代原调用进程的数据段、代码段和堆栈段，在执行完之后，原调用进程的内容除了进程号外，其他全部被新程序的内容替换。

exec 函数族共有六个函数，如下表所示：

表 2 exec 函数族成员的函数语法

所需头文件	#include <unistd.h>
函数原型	int execl(const char *path, const char *arg, ...) int execv(const char *path, char *const argv[]) int execle(const char *path, const char *arg, ..., char *const envp[]) int execve(const char *path, char *const argv[], char *const envp[]) int execlp(const char *file, const char *arg, ...) int execvp(const char *file, char *const argv[])
函数返回值	-1：出错

对于 exec 函数族的第一个参数，传递的均为程序名，只不过 execl 系列函数是包含路径；第二个参数，execl 系列函数需要将每个命令行参数作为函数的参数进行传递；而 execv 系列函数将所有函数包装到一个数组中，然后把指针数组首地址作为参数传递给 execv 类函数。

事实上，这 6 个函数中真正的系统调用只有 execve，其他 5 个都是库函数，它们最终都会调用 execve 这个系统调用，调用关系如下图所示：

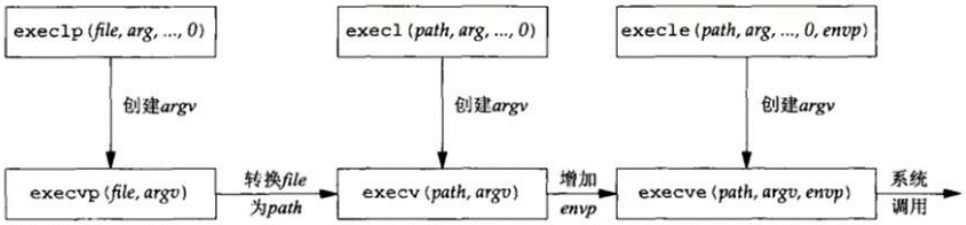

图 11 exec 函数组关系图

main 函数的返回值和系统功能调用 wait ()的关系

此处先给出我个人的理解：系统调用 wait(status)能获取子进程的退出状态，status 存储子进程 main 函数返回值。

main 函数的返回值用于说明程序的退出状态。如果返回 0，则代表程序正常退出。返回其它数字的含义则由系统决定。通常，返回非零代表程序异常退出。

wait 函数的功能是：父进程一旦调用了 wait 就立即阻塞自己，由 wait 自动分析是否当前进程的某个子进程已经退出，如果让它找到了这样一个已经变成僵尸的子进程，wait 就会收集这个子进程的信息，并把它彻底销毁后返回；如果没有找到这样一个子进程，wait 就会一直阻塞在这里，直到有一个出现为止。

wait()要与 fork()配套出现，如果在使用 fork()之前调用 wait()，wait()的返回值则为-1，正常情况下 wait()的返回值为子进程的 PID。

参数 status 用来保存被收集进程退出时的一些状态，它是一个指向 int 类型的指针。但如果对这个子进程是如何死掉毫不在意，只想把这个僵死进程消灭掉，可以设定这个参数为 NULL，即 pid = wait(NULL);。如果成功，wait 会返回被收集的子进程的进程 ID，如果调用进程没有子进程，调用就会失败，此时 wait 返回-1，同时 errno 被置为 ECHILD。

如果参数 status 的值不是 NULL，wait 就会把子进程退出时的状态取出并存入其中，这是一个整数值(int)，指出了子进程是正常退出还是被非正常结束的，以及正常结束时的返回值，或被哪一个信号结束的等信息。

可见系统调用 wait(status)能获取子进程的退出状态，status 存储子进程 main 函数返回值。由于这些信息被存放在一个整数的不同二进制位中，所以用常规的方法读取会非常麻烦，人们就设计了一套专门的宏(Macro)来完成这项工作，具体内容见下表，此内容来自刘福岩老师推荐的《UNIX 环境高级编程(第 3 版)》：

表 3 终止状态相关的四个宏

宏	说明
WIFEXITED(*status*)	若为正常终止子进程返回的状态，则为真。对于这种情况可执行 WEXITSTATUS(*status*)，获取子进程传送给 exit 或_exit 参数的低 8 位
WIFSIGNALED(*status*)	若为异常终止子进程返回的状态，则为真（接到一个不捕捉的信号）。对于这种情况，可执行 WTERMSIG(*status*)，获取使子进程终止的信号编号。另外，有些实现（非 Single UNIX Specification）定义宏 WCOREDUMP(*status*)，若已产生终止进程的 core 文件，则它返回真
WIFSTOPPED(*status*)	若为当前暂停子进程的返回的状态，则为真。对于这种情况，可执行 WSTOPSIG(*status*)，获取使子进程暂停的信号编号
WIFCONTINUED(*status*)	若在作业控制暂停后已经继续的子进程返回了状态，则为真（POSIX.1 的 XSI 扩展；仅用于 waitpid）

main 函数入口参数与返回值和 shell 语言中内置变量的关系

内置变量是程序预先定义好的一类特殊变量，用户只能使用内置变量，而不能创建新的内置变量，也不能直接为内置变量赋值。内置变量使用“$”符号和另一个符号组合表示，Shell 语言中常用内置变量如下：

- $$：表示当前运行脚本的进程 ID 号
- $!：表示后台运行的一个进程的 ID 号
- $@：与$#相同，但是使用时加引号，并在引号中返回每一个参数
- $-：显示 shell 使用的当前选项，与 set 命令相同
- $?：显示最后命令运行的推出状态，0 表示没有错误，其他任何值表示有错误

通过资料查询和实际操作后，可以得到如下关系：

- $#：即 argc，也就是入口参数个数
- $@：即 argv，也就是 main 函数的所有入口参数
- $? ：即 return，也就是 main 函数返回值

测试程序

```
#!/bin/bash
echo "脚本名：$0"
echo "第一个参数：$1"
echo "第二个参数：$2"
echo "第三个参数：$3"
echo "第四个参数：$4"
echo "第五个参数：$5"
echo "第六个参数：$6"
echo "第七个参数：$7"
echo "第八个参数：$8"
echo "第九个参数：$9"
echo "第十个参数：$10"
echo "第十个参数：${10}"
```

运行结果

```
yanxinyu@Thinkbook16-2022:~$ ./prog1.sh i l o v e s h u 1 1
脚本名：./prog1.sh
第一个参数：i
第二个参数：l
第三个参数：o
第四个参数：v
第五个参数：e
第六个参数：s
第七个参数：h
第八个参数：u
第九个参数：1
第十个参数：i0
第十个参数：1
yanxinyu@Thinkbook16-2022:~$
```

可以看到，${10}正确读取到了第十个参数，而$10 被分成$1，读取到第一个参数 i 然后拼接字符串 0，因此输出 i0。

(3) 请写出一个你上机时使用了上述五个系统功能调用的程序，简述程序的功能，简述你亲自调试该程序时遇到了哪些问题。（5 分）请不要抄袭，你写的程序不可能和别人的是一样的！

使用了上述五个系统功能调用的程序

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

int main() {
    int i, j;
    if (fork()) {
        i = wait(&j);
        printf("It is parent process.\n");
        printf("The child process,ID number %d,is finished.\n", i);
        execlp("ls", "ls", NULL);
    } else {
        int k = getpid();
        printf("It is child process.\n");
        sleep(10);
        exit(0);
    }
    return 0;
}
```

运行结果

```
yanxinyu@Thinkbook16-2022:~$ ./pro.o
It is child process.
It is parent process.
The child process,ID number 4378,is finished.
公共的  下载    f5          prog10.o   prog3.2.o  prog4.c  prog6.o   prog8.o
模板    音乐    filea       prog1.sh   prog3.3.c  prog4.o  prog7.o   prog9.c
视频    桌面    filedir.txt prog3.1.c  prog3.3.o  prog5.c  prog8a.c  prog9.o
图片    cppdir  pro.c       prog3.1.o  prog3.4.c  prog5.o  prog8a.o  pro.o
文档    f       prog10.c    prog3.2.c  prog3.4.o  prog6.c  prog8.c   snap
```

程序的功能

程序的主要功能是验证 fork 和 execlp 的功能，一开始就调用一个 fork()在主进程部分我们使用 wait 让子进程先运行，待子进程结束后输出信息，然后调用 execlp 调用 ls 展示列表。子进程则调用 getpid 展示子进程进程号。最后 exit 结束子进程。

此程序一开始无法正常运行，经过资料查询后我发现，我在使用 execlp 时没有带上最后的 NULL，修改后就可以正常运行了。

execlp 的函数详细如下：

```
int execlp(const char *file, const char *arg, ...);
```

execlp()会从 PATH 环境变量所指的目录中查找符合参数 file 的文件名，找到后便执行该文件，然后将第二个以后的参数当做该文件的 argv[0],argv[1], ...，最后一个参数必须用空指针(NULL)作结束。

(4) 写一个不少于 200 字的课程总结，包括课程体会，以及对课程教学的建议。（5 分）请不要抄袭，你的课程体会不可能和别人的是一样的！

个人认为，操作系统是计算机专业所有课程中最重要的一门。因为在本课程中，通过应用离散数学、数据结构、计算机组成原理的知识，可以利用操作系统对硬件系统进行首次扩充，管理好硬件设备，提高其利用率和系统的吞吐量，并为用户和应用软件提供了一个简单的接口，便于用户的使用，这为计算机领域的一切发展奠定了基础。

有人可能会问，学习这门课程有什么用呢？在我看来，对于芯片开发或者系统级部件开发而言，这部分知识可以为我们提供硬件与软件交互的基本理论支持，因为操作系统是配置在计算机硬件上的第一层软件。同时，该门课程也是整个计算机的底层基础，掌握好这门课程对于我们从事计算机软硬件方面的工作非常重要。即使将来从事纯软件开发，如果了解底层的运行原理、工作逻辑，可以设计出更适配的高质量代码，提升效率。

更重要的是，通过本课程的学习，可以掌握不少学习的方法。在操作系统(1)课程学习过程中，通过同学间互相探讨，我逐步提高了发现问题、分析问题和解决问题的能力；通过课程实验，提高了资料查找、相互合作、沟通交流的能力；同时，通过撰写这篇报告，我也提高了文字编辑、排版等方面的能力。

普通人的记忆力是有限制的。大学本科四年学习的专业课，在多年后回想其中的知识点，可能是模糊不清的。但是在学习操作系统(1)课程(甚至其他专业课)中掌握的学习技巧、方法，会流淌在血液中，帮助我们提高各方面的素养，可以更快的重新拾起遗忘的知识，更好的学习其他知识。我想这正是学习的意义所在。

当然操作系统(1)临近尾声，我想对该课程的授课形式、实验内容提出自己的建议。从一名学生的角度来看，我觉得操作系统(1)中所教授的部分内容过于古老，以至于和现代操作系统的差异很大。当然，掌握早期和经典的操作系统模型有利于入门，更方便学习和理解。因此，我想课程的讲授过程中，可以穿插一些研讨环节，让同学自行查找最新资料，介绍一些现代操作系统的相关内容，例如各个章节内容在现代操作系统中的实现和改进，并通过小实验的形式向大家展现，以达到与时俱进、拓宽视野的作用。

最后，我也想在这里感谢陈老师通过风趣、幽默、认真地教授，使我掌握了课程的知识点，并在课后耐心地解答我的困惑。感谢同学们对我的帮助。在大家的帮助下，顺利地完成了操作系统(1)课程的学习，期待冬季学期操作系统(2)的学习！

<报告结束>