**北京工业大学2022—2023学年第一学期**

**《操作系统》期末大作业 答题纸**

**考试说明： 开卷；14：00-18：00；倘若出现雷同卷，均按0分计成绩**

**姓名： 常兴阳 学号： 20080309 班号： 200803**

**。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。。**

**注：**本作业共 四 大题，共 7 页，满分100分。回答问题时，请在试题卷后附带的统一答题纸上作答，一定要标明对应的题号；不够的话，请自行续页。

**成 绩 汇 总 表（阅卷教师填写）**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **题号** | 一 | 二 | 三 | 四 |  |  |  |  |  |  | **总成绩** |
| **满分** | 10 | 30 | 35 | 25 |  |  |  |  |  |  |  |
| **得分** |  |  |  |  |  |  |  |  |  |  |

非常重要：

开卷；14：00-18：00；倘若出现雷同卷，均按0分计成绩

答题纸要在日新平台上提交，文件名为：学号+姓名

**大作业答题纸**

**姓名： 常兴阳 学号： 20080309**

**大作业答题纸**

一、基础应用题

1. 操作系统是管理计算机硬件、协调硬件资源并提供其他软件沟通平台的计算机软件程序

2．进程管理、内存管理、文件系统、网络通信、安全机制、用户界面、驱动程序。

进程管理：操作系统调整多个进程的功能，包括进程间通信、分时调度、死锁侦测等；

内存管理：虚拟内存管理、内存段页结合多级页表、内存换入换出、内存释放回收等；

文件系统：文件存储逻辑、存储方式、物理硬盘管理、错误恢复等；

网络通信：操作主流网络通信协议、网络收发交互等；

安全机制：认证资源访问、等级授权、资源访问日志、防火墙等；

驱动程序：与硬件间或硬件与软件间的操作软件；

用户界面：GUI、图形方式显示的计算机操作用户界面

二、基础应用题

1. 进程是计算机中已执行的程序。程序本身只是指令、数据及其形式的描述，进程是程序的真正运行实例。

进程概念的提出：批处理操作系统只允许一个程序在运行，内存中只有一个程序，无法满足人们需求，推出进程概念，表示应用程序在内存中分配的空间，且相互之间互不干扰，有助于有效管理调度内存中的程序。

进程状态包括 新建态、就绪态、运行态、堵塞态、终止态。

其中不可能的进程转化如表中×标记所示

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 终点\起始 | 新建态 | 就绪态 | 运行态 | 堵塞态 | 终止态 |
| 新建态 |  | × | × | × | × |
| 就绪态 | √ |  | √ | √ | × |
| 运行态 | × | √ |  | × | × |
| 堵塞态 | × | × | √ |  | × |
| 终止态 | × | × | √ | × |  |

2．**相似处**：

拥有独立的堆栈空间；

拥有独立的ID、状态和优先权；

均可竞争资源调度；

**不同处**：

进程拥有独立的数据段，有独立的地址空间和I/O资源，线程间共享数据段并使用相同的地址空间；

线程是独立调度和分派的基本单位，进程是资源分配和程序运行的基本单位，一个进程可以包括多个线程；

进程间数据共享复杂但易于同步，线程间数据共享简单但同步复杂；

进程开销较大，线程开销较小，切换较快；

进程之间独立，安全性较高，线程的问题出现会导致进程出问题；

3．用户级线程的整个过程不需要内核参与，于用户空间中实现，易于控制，但每个用户线程不具有自身的线程上下文，同一进程中只能同时有一个用户级线程在运行；核心级线程调度由内核负责，可在全系统内进行资源竞争，能更好利用多核CPU提高运行效率，一个进程的多个线程可以同时执行。

4.

(a)：不可行，左侧线程执行A()，到B()时，跳转倒B()将104地址压栈，跳转到200地址，运行到Yield()时，线程发生切换，204压入用户栈，CPU保护倒进程1的TCB中，并根据线程2的TCB恢复用户运行状态。右侧程序300地址开始执行，开始执行C()，运行到D()时，将304压栈，程序跳转到400。运行到Yield(),发生线程切换，404压入用户栈， CPU保护进程2到2的TCB中，根据线程1的TCB恢复现场。并继续执行。

(b)： void yield( ){

TCB2.esp = esp; // 当前线程的栈顶指针被存储到TCB

esp = TCP1.esp; //新线程的栈顶指针加载到CPU

}

三、综合应用题

1. 处理时会有进程间相互协作的情况，二者(或多者)之间有共享的数据需要传递同步，这便引出了进程同步的概念，即有序操作同步，保证数据一致性；在多组有相似功能的进程协同操作时，如多个生产者进程和多个消费者进程的情况时，信号量用来表示系统中某种资源的数量，用于资源的互斥访问，进而可唤醒或睡眠调度相应的多个进程；临界区是修改共同变量的对应代码段，特 点是该代码段在同一时间只允许一个进程同时执行，避免了多个进程同时修改信号量导致的差错。总结则是：进程同步是目的，信号量是资源保护的机制，临界区是保证同步数据的区域不会同时被多个进程修改的手段，保护信号量。

2. 上述程序在运行过程中会发生死锁。分析阐述如下：当缓冲区中商品数量为空时，消费者执行时，先进行加锁测试，加锁成功表示其他尝试加锁的程序被堵塞，即此时生产者程序被堵塞于P(&mutex)；再进行P(&full)的测试，查看缓冲区中是否有商品供消费，由于此时商品数量为空，所以消费者会带着锁堵塞在P(&full)处，就会产生死锁的情况；消费者的堵塞状态需要生产者进入临界区增加商品来解除，生产者的堵塞状态需要消费者解除临界区的加锁才能进入临界区，但消费者临界区的锁解除需要在消费者完成消费过程才能被解除，逻辑依赖闭环，这就导致了死锁的产生。

此外，当商品数量不为0时，生产者由于P(&empty)发生的堵塞和进入临界区后引发的消费者由于P(&mutex)发生的堵塞不会引发死锁。

3. **判断方法**：

前提判断：若系统当前可用资源数量向量无法满足任意一资源等待矩阵中行向量的值，或存在进程对资源的需求总量超过系统资源总量，系统会发生死锁。

Ⅰ.寻找一个没有标记的进程，它于资源等待矩阵中的对应的行向量中每一个值都小于系统当前可用资源数量向量，说明该进程请求的资源可以被满足，该进程可以被运行。

Ⅱ.如果能找到这样的进程，将其打上标记，将资源分配矩矩阵的该进程行向量加到系统当前可用资源数量向量中，此行为表示该进程运行完，释放所占有的资源，并被回收至系统可用资源数量。

Ⅲ.如果没有这样的进程，那么算法终止。

Ⅳ.算法终止时若存在没有标记的进程，说明此进程处理顺序不能完成所有进程；若所有的进程处理顺序均不能完成所有进程，则说明该例子会发生死锁。

**死锁判断分析：**结论为该例子会发生死锁。分析如下：按照上文所述判断方法中的前提判断，先判断是否发生“系统当前可用资源数量向量无法满足任意一资源等待矩阵中行向量的值”的情况。系统当前**可用资源2**数量不足以满足**进程2对资源2的需求**，系统当前**可用资源3**数量不足以满足**进程1、3对资源3的需求**，系统当前**可用资源5**数量不足以满足**进程3、4、5对资源5的需求**，综上，系统当前可用资源不足以满足任意一个进程对资源的需求，例子中的所有进程均无法被运行，发生死锁。

四、综合论述题

否定之否定规律是哲学的基本规律之一。 它揭示了事物发展的**前进性与曲折性的统一**，表明了事物的发展不是直线式前进而是**螺旋式上升**的。

内存管理的措施并不是最初便得出段页混合的方式的，从无到现在所使用的段页混合经历了多次的尝试和改进，是典型的理工科中“实践🡪发现问题🡪修改优化🡪实践🡪再发现问题”螺旋的情况。由于先前的批操作操作系统中未涉及多进程的概念和实现，内存的使用和管理简单粗暴，即**固定加载地址**，即唯一运行的程序可以且仅可以加载在在内存中固定的位置才能正常运行，但由于只有这一个程序运行，倒也没什么问题。当人们开始制作可以多进程运行的操作系统时，这个方法就严重不适用，科学家们最初采用的内存管理措施是**固定分区**，这是人们最容易想到的方法。将内存空间平均分为若干相同大小的空间，一个程序要运行就给他分配这样一个空间，但是这样简单的分配方法缺点很明显，一个过大或过小的程序都不适用：过小的程序只使用到了分配内存的极小一部分，大部分分配的都在浪费且不能用于它图；过大的程序无法在完整加载于分配的内存大小空间内，受此影响，**非固定分区**被改进出来：一个程序就在内存空间中就是完整的一大段连续空间，多个程序在内存中就是多个连续的大段空间，由于内存分割不再整齐，为了区分找到各个程序的起始位置，使用了**段表**存储程序和程序的起始地址。人们对同时处理的进程数量要求逐渐增高，内存大小难以满足人这方面的需求，但是硬盘空间比较充裕，**交换**技术应运而生,将不活跃的进程存储于硬盘中，再运行时再从硬盘中加载到内存，变相解决了内存大小不够的问题。

随着使用，内存纯粹分段的劣势展现了出来：按需分配内存大小是方便首次运行应用需要了，但是大大小小程序释放后产生的不连续、大小不一的内存空间难以整理用于下一个程序的使用，很难每次都找到一个完美匹配新程序的内存空间的段，而内存重整型大块内存移动十分影响性能和使用体验，故需要新的策略解决此问题。**分页**就是接替纯粹分段出现的。它将内存整齐地划分成小块，同时将程序也按页大小且成多个固定大小的页面，一个程序分别连续地存放于多个页中，既保证了程序一定程度上的连续，又保证回收得到的内存空间大小可适用于接下来的。页的划分很小，段表也升级成了**页表**进行管理，所有的页都对应的是哪些程序。划分的小带来了内存空间较高的利用率和灵活的存取，同时也带来了大量页表存储读取查询的困难，表中的大多条并不会在运行时被用到，使用多级页表和块表才逐渐解决页表大量资源占用的问题。

当然，分页并不是完美的解决方案，分页或许解决了当时一段时间的处境，但随着软件规模的增大，人们对计算机性能的更高需求，纯分页管理的不足逐渐显露出来：不方便按照逻辑模块实现信息的共享和保护，因为大型应用在内存中被切分得太细碎了，某个应用的数据模块或逻辑模块可能并未能完整加载到内存空间中，且定长的数据模块和会发生内存需求改变的逻辑模块被放置到一起，而完整加载进内存的模块又不一定被当前使用，这大大影响了软件的运行效率，于是“分段”被重新拾起，应用按照作用划分为数据段、逻辑划分为几段加载并按照特性和需求储存到不同的区域，比如定大小不可修改的数据段被放置在一起，会发生大小改变的程序段则排在一起，这便是**逻辑分段**，既然是段式管理**段表**则也是不可或缺的记录方式。

当然这还不是最终方案，分段的缺点在上方已经介绍过一遍了，纯分段方式会带来大块不便的内存需求以及内存碎片的产生，最终，结合段式管理和页式管理的优缺点，**段页式**内存管理成为了内存管理的接班人，程序按照逻辑分为多端，按需加载定长的数据模块和可能变化的逻辑模块，避免了全部载入内存但并非所有都使用的浪费；每一段内为了便于释放后的内存回收进行了分页，集各种方法的优点，尽量避免了管理方法的缺点，经历这么多次的技术迭代，最终得到了使用**段页表**来管理的**段页式**内存管理方案