Press release

CSOAsupport
Nacky学长
+8615913510179
service@csoasupport.com

FOR Operating System

计算机操作系统原理考试题 | OS Principles Assignment

OS 代写 | 操作系统作为代做 | 系统设计代做

(i) [5 分] 在一个计算机系统中，主存储器中同时运行着六个独立的程序。已知每个程序在执行过程中，有四分之一的时间处于因等待输入/输出操作（I/O）完成而造成的空闲状态。请问，CPU 时间的浪费比例是多少？

答案：根据题意，每个程序因等待 I/O 而处于空闲状态的概率为 1/4。要计算所有这六个程序在同一时刻都处于等待 I/O 的空闲状态的概率，我们需要计算每

个程序空闲概率的乘积，即 (1/4)^6。计算结果为 1/4096。这意味着，在 4096 个时间单位中，有 1 个时间单位所有程序都处于空闲状态，此时没有程序可以供中央处理器（CPU）执行，导致 CPU 处于空闲状态。这部分 CPU 时间未能得到有效利用，因此被视为浪费。

评分标准：能够提及单个进程因等待 I/O 而空闲的概率，可获得 2 分；能够正确计算出所有六个进程同时处于空闲状态的概率，可获得 4 分；给出完全正确的答案，可获得满分 5 分。

(ii) [8 分] 当发生中断事件或用户程序（进程）发起系统调用请求时，控制权会从用户态程序转移到操作系统内核态。在这个过程中，操作系统会使用一个专门的内核栈，这个内核栈与被中断的用户进程所使用的用户栈是相互独立的。请解释为什么在这种情况下需要使用内核栈，而不是直接使用用户栈？

答案：当中断或系统调用发生，需要将控制权从用户程序（进程）转移到操作系统时，使用内核栈的主要目的是为了安全地保存被中断进程的上下文数据，以便在系统调用处理完成后能够正确地恢复用户程序的执行。此外，系统调用陷入内核后，需要执行一些特权指令，这些指令的操作也需要在内核栈上进行。内核栈由操作系统的调度器和分派器等内核程序使用。之所以不能使用用户栈，是因为用户栈位于用户地址空间，其内容可能被恶意的用户进程有意篡改，或者被用户进程自身的错误操作无意中破坏。内核栈则受到操作系统的严格保护，仅能由内核代码进行修改，这样可以确保其数据的完整性和安全性，防止因用户程序的异常行为导致系统崩溃或数据泄露。

评分标准：能够提及在控制权转移时使用内核栈的原因，可获得 4 分；能够解释为什么用户栈不适合在这种情况下使用，可获得 4 分。

(iii) 给定一段包含进程创建的代码（原题中未提供，此处假设代码会创建子进程），请问屏幕上会打印 "printf statement - i is <number>" 这条语句多少次？打印出的 i 值分别是多少？请通过提供一个清晰的进程树结构，并在每个进程节点上标注其对应的 i 值，来详细解释你的答案。

答案：假设主进程（我们称之为 P0）首先创建了两个子进程，分别标记为 P1 和 P2。然后，假设子进程 P1 又创建了一个新的子进程，我们标记为 P3。这样，总共有四个进程在运行，包括最初的主进程。因此，代码中的 `printf` 语句会被执行四次，每个进程执行一次。

现在我们来分析 i 值的变化。假设主进程 P0 在创建子进程之前，其变量 i 的初始值为 3，并且在创建子进程之后，主进程 P0 将 i 的值增加了 3。因此，主进程 P0 最终打印出的 i 值为 6。

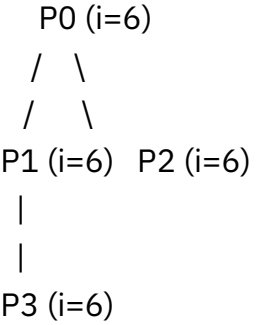
当主进程 P0 使用 `fork()` 系统调用创建子进程 P1 时，P1 会继承父进程 P0 在 fork 时刻的 i 值，即 3。然后，P1 将其 i 值增加了 3，所以 P1 最终打印出的 i 值为 6。

同样地，当主进程 P0 创建子进程 P2 时，P2 也会继承父进程 P0 在 fork 时刻的 i 值。需要注意的是，在创建 P2 之前，P0 的 i 值已经被更新为 6。因此，P2 在 fork 时获得的 i 值为 6，并且 P2 没有再修改 i 的值，所以 P2 最终打印出的 i 值为 6。

最后，当子进程 P1 创建子进程 P3 时，P3 会继承父进程 P1 在 fork 时刻的 i 值，即 6。P3 也没有再修改 i 的值，所以 P3 最终打印出的 i 值为 6。

综上所述，屏幕上会打印 "printf statement - i is <number>" 四次，打印出的 i 值分别为 6, 6, 6, 6。

进程树结构如下：



2 (20 分)

(i) [6 分] 假设你需要设计一种中央处理器（CPU）调度算法，该算法的核心思想是优先调度那些在最近一段时间内使用处理器时间最少的进程。请详细阐述你将如何开发这种算法，以确保能够优先处理输入/输出密集型（I/O-bound）程序，同时避免出现计算密集型（CPU-bound）程序长时间无法获得 CPU 资源而导致的“饥饿”现象。在解释你的算法时，请明确指出你所做的所有假设，例如所使用的数据结构、表格或变量等。

答案：我将设计一种基于历史 CPU 使用时间的优先级调度算法，并结合老化机制来防止 CPU 密集型程序饥饿。

算法描述：

1. 数据结构：
进程控制块 (PCB)：每个进程都有一个 PCB，其中包含进程的基本信息，以及以下用于调度的关键字段：
进程 ID (PID)
当前状态 (就绪、运行、阻塞等)
最近 CPU 突发时间 (RecentCPUBurstTime)：用于记录进程最近一次或几次连续使用 CPU 的时间长度。我们可以采用指数衰减平均法（Exponential Averaging）来计算这个值，即新的 RecentCPUBurstTime = α * 上一次 CPU 突发时间 + (1 - α) * 前一个 RecentCPUBurstTime，其中 α 是一个介于 0 和 1 之间的权重因子，用于控制历史数据的影响程度。
优先级 (Priority)：进程的优先级将根据其 RecentCPUBurstTime 来动态调整。
等待时间 (WaitingTime)：记录进程在就绪队列中等待的时间。
就绪队列 (Ready Queue)：一个按进程优先级排序的队列，用于存放所有处于就绪状态的进程。

2. 算法步骤：

当一个进程进入就绪队列时，或者当一个正在运行的进程完成其 CPU 突发或发生 I/O 请求而进入阻塞状态时，调度器会被触发。
调度器遍历就绪队列中的所有进程，并根据它们的 RecentCPUBurstTime 计算或更新其优先级。优先级分配策略是：RecentCPUBurstTime 越短的进程，其优先级越高（例如，可以使用一个数值表示优先级，数值越小表示优先级越高）。
调度器选择就绪队列中优先级最高的进程进行调度，并将其状态设置为“运行”。
当一个进程完成一个 CPU 突发并需要进行 I/O 操作时，它的 RecentCPUBurstTime 会被记录下来，并用于更新其 PCB 中的 RecentCPUBurstTime。然后，该进程进入相应的 I/O 等待队列。
当一个进程完成 I/O 操作并返回就绪队列时，它的 RecentCPUBurstTime 可能会被重置或以某种方式更新，然后根据其当前的 RecentCPUBurstTime 重新计算优先级。

3. 偏向 I/O 密集型程序：由于 I/O 密集型程序通常具有较短的 CPU 突发时间，因此它们的 RecentCPUBurstTime 会相对较小，从而获得更高的优先级，更容易被调度执行。

4. 防止 CPU 密集型程序饥饿（老化机制）：为了防止 CPU 密集型程序因为长时间得不到 CPU 而发生饥饿，我们引入老化机制：

基于等待时间的优先级提升：可以定期检查就绪队列中进程的等待时间。如果一个进程在就绪队列中等待的时间超过某个阈值，就逐渐提高其优先级（例如，减小其优先级数值）。这样，即使一个进程的 RecentCPUBurstTime 较长，但只要它等待的时间足够长，其优先级也会逐渐提高，最终获得执行机会。
定期调整优先级基准：可以定期调整计算优先级的基准，例如，在一定时间间隔后，对所有进程的 RecentCPUBurstTime 进行一定的调整，使得长时间未运行的 CPU 密集型程序的优先级能够得到提升。

假设：

我们能够准确地测量每个进程的 CPU 突发时间。
我们维护了一个就绪队列，并且能够根据进程的优先级有效地进行排序和选择。
老化机制的参数（如等待时间阈值、优先级提升幅度、优先级调整周期等）需要根据系统的具体情况进行调整和优化。

(ii) [6 分] 轮询（RR）调度算法为每个进程分配一个固定的时间片来执行。如果进程在时间片用完之前没有主动放弃 CPU，则会被强制剥夺 CPU 的使用权（即发生抢占），然后将 CPU 分配给就绪队列中的下一个进程，也执行一个时间片。现在考虑有三个进程 P1、P2 和 P3 按照给定的顺序到达系统，它们的到达时间均为 0 毫秒，每个进程的 CPU 突发时间均为 12 毫秒。轮询调度算法的时间片设置为 2 毫秒，上下文切换的时间为 1 毫秒。请计算出每个进程的周转时间和等待时间，并展示你的计算过程，最后将答案填入表 1 中。

计算过程：

时间 (ms)	运行进程	就绪队列 (尾部)
0	P1	P2, P3
2	P2	P3, P1
4	P3	P1, P2
6	P1	P2, P3
8	P2	P3, P1
10	P3	P1, P2
12	P1	P2, P3
14	P2	P3, P1
16	P3	P1, P2
18	P1	P2, P3
20	P2	P3, P1
22	P3	P1

24	P1	
26	P2	
28	P3	

详细时间线：

* **0-2 ms:** P1 运行 (2 ms)，剩余 10 ms。就绪队列：P2, P3。上下文切换 (1 ms)。
* **3-5 ms:** P2 运行 (2 ms)，剩余 10 ms。就绪队列：P3, P1。上下文切换 (1 ms)。
* **6-8 ms:** P3 运行 (2 ms)，剩余 10 ms。就绪队列：P1, P2。上下文切换 (1 ms)。
* **9-11 ms:** P1 运行 (2 ms)，剩余 8 ms。就绪队列：P2, P3。上下文切换 (1 ms)。
* **12-14 ms:** P2 运行 (2 ms)，剩余 8 ms。就绪队列：P3, P1。上下文切换 (1 ms)。
* **15-17 ms:** P3 运行 (2 ms)，剩余 8 ms。就绪队列：P1, P2。上下文切换 (1 ms)。
* **18-20 ms:** P1 运行 (2 ms)，剩余 6 ms。就绪队列：P2, P3。上下文切换 (1 ms)。
* **21-23 ms:** P2 运行 (2 ms)，剩余 6 ms。就绪队列：P3, P1。上下文切换 (1 ms)。
* **24-26 ms:** P3 运行 (2 ms)，剩余 6 ms。就绪队列：P1, P2。上下文切换 (1 ms)。
* **27-29 ms:** P1 运行 (2 ms)，剩余 4 ms。就绪队列：P2, P3。上下文切换 (1 ms)。
* **30-32 ms:** P2 运行 (2 ms)，剩余 4 ms。就绪队列：P3, P1。上下文切换 (1 ms)。
* **33-35 ms:** P3 运行 (2 ms)，剩余 4 ms。就绪队列：P1, P2。上下文切换 (1 ms)。
* **36-38 ms:** P1 运行 (2 ms)，剩余 2 ms。就绪队列：P2, P3。上下文切换 (1 ms)。
* **39-41 ms:** P2 运行 (2 ms)，剩余 2 ms。就绪队列：P3, P1。上下文切换 (1 ms)。
* **42-44 ms:** P3 运行 (2 ms)，剩余 2 ms。就绪队列：P1, P2。上下文切换 (1 ms)。
* **45-47 ms:** P1 运行 (2 ms)，完成。
* **47-49 ms:** P2 运行 (2 ms)，完成。
* **49-51 ms:** P3 运行 (2 ms)，完成。

完成时间：

P1 完成于 47 ms。
P2 完成于 49 ms。
P3 完成于 51 ms。

周转时间 (Turnaround Time) = 完成时间 - 到达时间：

P1 周转时间 = 47 - 0 = 47 ms。
P2 周转时间 = 49 - 0 = 49 ms。
P3 周转时间 = 51 - 0 = 51 ms。

等待时间 (Waiting Time) = 周转时间 - 突发时间：

P1 等待时间 = 47 - 12 = 35 ms。
P2 等待时间 = 49 - 12 = 37 ms。
P3 等待时间 = 51 - 12 = 39 ms。

表 1：

进程	周转时间 (ms)	等待时间 (ms)
P1	47	35
P2	49	37
P3	51	39

(iii) [8 分] 作为一名系统软件开发人员，你现在面临一项任务，需要开发一个互联网浏览器应用程序，该应用程序允许用户同时打开多个不同的标签页。你会选择将每个新的标签页实现在一个新的操作系统进程中，还是在一个已有的进程中创建新的线程来管理？请阐述你选择的方法相比另一种方法的优点和缺点。

答案：我会选择在同一个浏览器进程中为每个新的标签页创建一个独立的线程。

优点：

资源效率更高：创建线程比创建进程的系统开销要小得多。线程共享父进程的内存空间和许多其他资源（如文件句柄），因此创建和销毁线程的速度更快，占用

的系统资源也更少。这对于需要频繁创建和销毁标签页的浏览器应用来说至关重要。
上下文切换更快：线程之间的上下文切换通常比进程之间的上下文切换更快，因为线程共享相同的地址空间，不需要切换内存管理单元（MMU）等。这可以提高浏览器的响应速度和整体性能。
更方便的数据共享和通信：同一个进程内的多个线程可以直接共享内存中的数据，进行通信也更加方便和高效。例如，浏览器主进程可以方便地与各个标签页线程共享配置信息、缓存数据等。

缺点：

安全性较低：由于同一个进程内的所有线程共享相同的地址空间，如果一个线程崩溃或出现安全漏洞，可能会影响到整个浏览器进程，甚至导致所有标签页都崩溃。相比之下，如果每个标签页都在独立的进程中运行，一个标签页的崩溃通常不会影响到其他标签页。
同步和并发控制更复杂：由于多个线程共享相同的内存空间，因此需要更加谨慎地处理线程之间的同步和并发控制，以避免出现数据竞争、死锁等问题。这增加了应用程序开发的复杂性。

与基于进程的方法相比：

基于进程的方法的优点：

更高的安全性：每个标签页都在独立的进程中运行，拥有独立的地址空间，一个标签页的崩溃不会影响到其他标签页，安全性更高。
更好的隔离性：不同标签页之间相互隔离，一个标签页的恶意行为或错误不会直接影响到其他标签页。

基于进程的方法的缺点：

更高的资源开销：创建和管理多个进程需要更多的系统资源，包括内存和处理器时间。
更慢的上下文切换：进程之间的上下文切换开销较大。
进程间通信（IPC）更复杂：不同进程之间的数据共享和通信需要使用专门的进程间通信机制，例如管道、套接字、共享内存等，实现起来更加复杂。