Schedlab 实验报告

郭丹琪 2018202067

一.调度算法思路

为需要使用 CPU 资源和需要进行 IO 操作的任务各设一条队列，任何一个任务都只能在其中一条队列中。当需要选择任务进行 IO 操作的时候，选 IO 任务等待队列里截止时间最近且还未超时的任务。选择任务使用 CPU 资源的时候，同样选择 CPU 任务等待队列里截止时间最近且还未超时的任务。

二.实验过的调度算法分析

第一版算法

算法描述：

为等待 CPU 资源和 IO 资源的任务各设置两个等待队列，分别为高优先级任务的等待队列和低优先级任务的等待队列，则一共有四个等待队列。该队列是以 map 数据结构存储的，其中 key 值为任务的截止时间，value 值为任务的结构体，则根据 map 的特点，等待队列会自动按截止时间从近到远排列。

正在使用 CPU 资源或 IO 资源的任务不在等待队列中，另外存储。在调度任务的时候，如果 CPU 资源或 IO 资源正在被使用，则不进行抢占，资源空出来后再选择任务。

在选择任务的时候，选择截止时间最近且未超时的任务，且优先考虑高优先级队列，若高优先级队列为空才考虑低优先级队列。

算法问题：（1）非抢占式的调度可能会导致有的任务长时间无法运行；（2）如果 CPU 资源正在被使用时，有截止时间更近的新任务到达，则该新任务只能等当前任务结束后才能使用 CPU 资源，可能错过截止时间；（3）如果优先级低的任务的截止时间都很近而优先级高的任务截止时间很远，则会导致优先级低的任务全错过截止时间；（4）没有区分 CPU 密集型任务和 IO 密集型任务的情况。

算法分数：55

第二版算法

算法描述：

在第一版算法的基础上加入了抢占的操作。在高优先级任务的等待队列里找到截止

时间最近且还未超时的任务，若该任务比正在运行的任务截止时间近，则进行抢占。如果高优先级等待队列里无任务或任务都超时，则在低优先级等待队列找截止时间最近且未超时的任务，如果当前正在运行的任务超时或者当前任务是低优先级任务且截止时间更远，则进行抢占。

算法问题：（1）因为以高优先级的任务优先，如果优先级低的任务的截止时间都很近而优先级高的任务截止时间很远，则会导致优先级低的任务全错过截止时间。（2）没有区分 CPU 密集型任务和 IO 密集型任务的情况

算法分数：80

第三版算法

算法描述：

因为感觉原先的算法不好改进，所以直接改变了原先的算法，通过计算权重来选择要运行哪个任务，这样改进的时候如果要考虑其他因素的时候就可以调整权重。这个权重是由截止时间和优先级得出的，且由于 map 数据结构的特点，再加上截止时间是越小越优先，所以规定是权值越低越优先。因为在权值里同时考虑了截止时间和优先级，所以可以把原本要分优先级的等待队列合并成一个，写的时候思路会比原来的算法清晰。优先级高的优先级 prior 记为 0，优先级低的记为 0，权值 weight=截止时间*(prior+1)。在选择要运行的程序或者要进行 I/O 的程序时，对等待队列进行遍历，由于等待队列中的任务排得越前面越优先，找到第一个未超过截止时间的任务运行即可。在抢占的时候，也是找到第一个未超过截止时间的任务，然后再与当前任务的权值比较，权值低的运行。

算法问题：测试点 13-16 的分数较低，因为权值设置的原因，还是会导致部分任务应该先做的任务权重低。且后续发现在区分 CPU 密集型任务和 IO 密集型任务的情况时分数会降低。

算法分数：82

第四版算法:

算法描述:

该算法即为最终算法。直接放弃对其他因素的考虑，只考虑截止时间，不计算权重，不考虑优先级，等待 CPU 资源和 IO 资源的任务都分别只有一条等待队列。任何一个任务都只能在其中一条队列中。当需要选择任务进行 IO 操作的时候，选 IO 任务等待队列里截止时间最近且还未超时的任务。选择任务使用 CPU 资源的时候，同样选择 CPU 任务等待队列里截止时间最近且还未超时的任务。

算法问题:没有考虑优先级因素，没有区分 CPU 密集型任务和 IO 密集型任务的情况。但

是因为该算法分数最高，所以就不再改进了。

算法分数:92

三.实验中遇到的问题和解决办法

在改进第二版算法的时候遇到瓶颈，在原有算法上难以加入对其他因素的考虑，所以最后换了一种算法来写。

在改进第三版算法的时候也遇到瓶颈，尝试加入对各种因素的考虑反而会让分数变低，最后询问了已经写完的同学。原本以为第一版算法已经是很简单的，考虑条件最少的，但没想到竟然只用考虑截止时间这一因素。所以写出了最简单的第四版算法，并且得到教训，以后哪怕第一版算法也要从最最简单的、只考虑一种因素的开始写。