Corey Talbert
Assignment 1 Report
09-19-2022
CS 433-01 Operating Systems
Xiaoyu Zhang

Assignment 1 Program Report: Ready Queue

Introduction to the Ready Queue

This project implements a *ready queue*, a data structure representing processes that are prepared to be executed by the CPU, but waiting for a core to be allocated to it. When a program is initiated, it becomes a process. The relevant information needed to run it, including the program code, program counter, register values, memory allocation, and more, is stored in a structure called the *process control block* (PCB). Besides the ready queue, this project also includes a simple PCB implementation, as well as a PCB table that holds all PCBs in any state. In all, these abstractions form an essential potion of the process scheduling system.

A CPU scheduler chooses among ready processes to give a CPU core to for execution. In this implementation, when a process is first created its state is NEW. When it is added to the ready queue the state is changed to READY. When the state is removed from the ready queue its state changes to RUNNING. In a working operating system, the running process may complete, be interrupted and forced to wait, or be terminated. A process may also be swapped from pool of waiting processes and saved to disk when main memory runs low. When a process is interrupted its state must be saved and a new task loaded in an operation called *context switching*. Because context switching happens very frequently the time efficiency of the various process queues and schedulers is critical.

Data Structure

Ready queues may be implemented as various data structures including heaps and linked lists. This project, however, uses a hash map, implemented as an array of linked lists. The hash map was chosen because of the relative simplicity of implementation (especially compared to heaps that are more realistically used for this purpose). The process's priority is used as the hash key. If the number of priority levels were not known or not finite, a more complicated hash function would be needed. Hash collisions are handled through separate chaining using a simple linked list, so that PCBs with the same priority are stored in the same bucket.

Time

If a PCB is added to the ready queue, it is simply appended to the linked list at the key implied by its priority, giving constant insertion time O(1).

When a process is requested from the ready queue, the head of the linked list at that key value is given. This results in first-in-first-out behavior, such that the oldest process is removed from the bucket first. (This may or may not be useful functionality.) Then, if the ready queue is not empty, it is searched sequentially for the next highest priority process. This search may add significant time to the access operation. The lookup time complexity then is O(1) in the best cases when the next process is in the same bucket as the one just removed (the bucket size is tracked), O(1) when the ready queue is empty after removing a process (the queue size is tracked), or linear time O(n) where n is the number of buckets (priority levels) in the worst case when the process removed is the maximum possible priority and the next ready process is the minimum possible priority. Because the highest priority process is

tracked by sequential search, the average lookup time is less than O(n) and approaches O(1) as the queue fills.

The ready queue as implemented is like an unsorted array, with gaps consisting of empty buckets between non-empty buckets. It may be possible to improve the average search time of finding the next job by storing the buckets in order, close together, at the next available array index and choosing a more efficient search algorithm. Insertion time would be increased, though, when buckets need to be resorted. The overall effect is uncertain, and depends on the frequency of insert and remove operations.

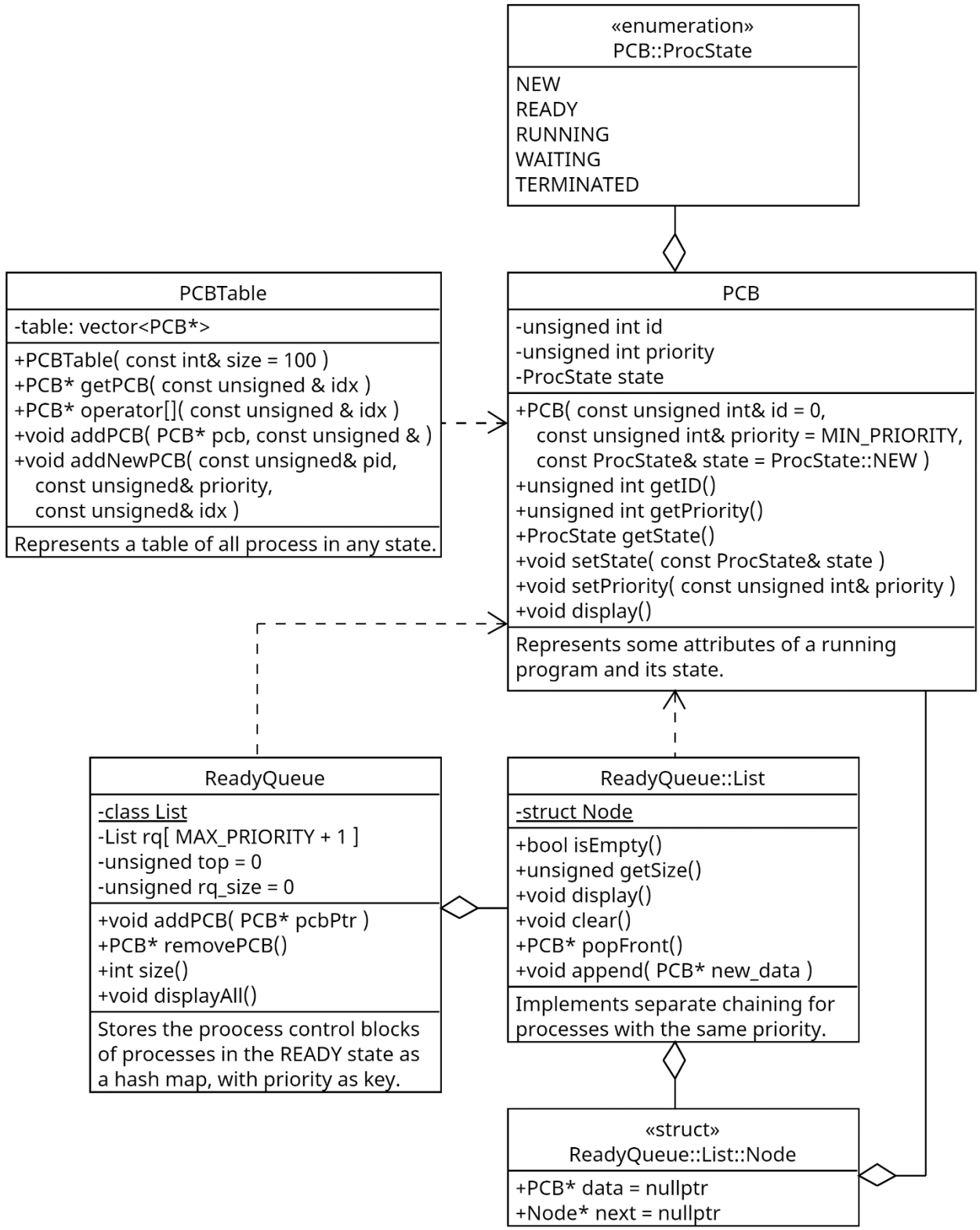
In testing, a non-optimized binary typically completes test2 in under 0.1 seconds, usually between 0.07 and 0.09 seconds. When complied with g++ optimization tag -O3, test2 finishes in about 0.05 seconds.

Space

The simple implementation and decent time complexity of the hash map comes at the cost of significant memory. The ready queue creates as many buckets as there are priority levels, or O(n) space complexity, where n is the number of priority levels. Introducing a hashing function, for example the modulo operation, may reduce the memory required at the cost of increasing lookup time. In this example, PCBs with different priorities such that priority % (some arbitrary denominator) yields the same key would be stored in the same bucket. Then the bucket could be searched of the right process. Using a second-level hash table instead of a linked list for separate chaining would reduce space without compromising the O(1) lookup time.

Corey Talbert
Assignment 1 Report
09-19-2022
CS 433-01 Operating Systems
Xiaoyu Zhang

UML Diagram



Corey Talbert
Assignment 1 Report
09-19-2022
CS 433-01 Operating Systems
Xiaoyu Zhang

Project files

- readyqueue.h
- readyqueue.cpp
- pcbtable.h
- pcbtalbel.cpp
- pcb.h