A Hybrid Lock-Aware Round Robin Scheduler to Reduce Wasted Quanta Under Critical-Section Contention

Ahmad Faiz
Dept. of Computer Science
SEECS, NUST
Islamabad, Pakistan
ahmad.bscs23seecs@seec.edu.pk

Arham Ali
Dept. of Computer Science
SEECS, NUST
Islamabad, Pakistan
a.alibscs23seecs@seec.edu.pk

Okasha Rehman
Dept. of Computer Science
SEECS, NUST
Islamabad, Pakistan
orehman.bscs23seecs@seec.edu.pk

***Abstract*—In typical operating systems courses, CPU scheduling and synchronization are often introduced as separate topics: scheduling algorithms such as Round Robin focus on fairness and response time, while synchronization primitives such as locks and semaphores focus on correctness and race-condition avoidance. In practice, however, these aspects interact in subtle ways. Under heavy contention for a shared lock, a standard Round Robin scheduler may repeatedly allocate CPU time to processes that immediately block on a locked critical section, effectively wasting quanta while delaying the process that can release the lock.**

This paper describes the design of a simple lock-aware extension to Round Robin scheduling. An initial approach based on penalizing processes that hold locks for long periods is shown to be counterproductive, because delaying the lock holder can increase the time for which other processes remain blocked. Motivated by this observation, a hybrid scheduler is proposed that preserves Round Robin behavior among processes that can make progress, while temporarily deferring processes that would only block on a currently held critical section. Deferred processes are placed in a secondary queue and reintroduced once the relevant lock is released. Conceptually, this combines Round Robin with a lightweight, dependency-based priority mechanism. The design aims to reduce wasted CPU time under contention and to shorten lock holding durations, while remaining simple enough for educational use.

***Index Terms*—Operating systems, CPU scheduling, Round Robin, synchronization, critical section, race condition, hybrid scheduling, lock-aware scheduling.**

I. Introduction

In introductory material, CPU scheduling algorithms such as Round Robin (RR) are typically presented as clean and fair time-sharing mechanisms: every ready process receives a time slice and starvation is avoided. Separately, race conditions, critical sections, semaphores, and classic problems such as producer–consumer and bounded buffer illustrate the need for concurrency control.

Whenever processes share data or resources, they must use synchronization primitives—such as mutex locks or semaphores—to protect critical sections and avoid race conditions [1], [2]. Correct synchronization prevents simultaneous access, but it also means that processes can be blocked while waiting for a lock. Standard schedulers, including Round Robin, simply see these processes as "blocked" or "ready"; they do not know that a ready process may be about to block immediately on a lock held by another process.

Consider a situation in which process P_1 is inside a critical section and holds a lock on a shared resource. Processes P_4 and P_5 are also in the ready queue and will soon need that same critical section. Under plain Round Robin, the scheduler still gives P_4 and P_5 their usual turns. When they run, they attempt to enter the critical section, find the lock held by P_1, and block immediately. The CPU time slice given to them does not allow useful work; it only moves them from ready to blocked. At the same time, P_1 may receive its next time slice later than necessary, keeping the lock longer and extending the blocking time of others.

This motivates the question of whether a scheduler can be modestly aware of lock-related dependencies, without significantly increasing complexity. This paper investigates a simple modification of Round Robin that takes such dependencies into account.

The main contributions are as follows:

- An initial history-based design is described, in which processes frequently blocked on locks are rewarded and processes that hold locks for long periods are penalized, together with an explanation of why this approach can unintentionally increase blocking times.
- A hybrid scheduler is proposed that preserves Round Robin behavior among processes that can make progress, while temporarily deferring processes that would immediately block on a currently held critical section.
- An informal analysis, with an illustrative example and pseudocode, is provided to explain how this hybrid approach can reduce wasted quanta under lock contention without changing the semantics of locks.

The rest of the paper is organized as follows. Section II reviews basic concepts in scheduling and synchronization. Section III explains the first history-based design and the problems encountered. Section IV presents the final hybrid design based on a main and secondary ready queue, including

pseudocode. Section V discusses an illustrative example and potential behavior. Section VI concludes the paper.

II. Background

A. CPU Scheduling and Round Robin

CPU scheduling is the task of selecting which ready process should run on the CPU next [1], [3]. Common goals include high CPU utilization, low waiting time, low turnaround time, and fairness among processes.

Round Robin is a classic preemptive scheduling algorithm used in time-sharing systems. It maintains a queue of ready processes and repeatedly:

1) Dequeues the first process.
2) Runs it for a fixed time quantum Q.
3) If the process finishes within Q, it terminates.
4) If not, it is preempted and placed at the end of the queue.

Because each ready process eventually reaches the front of the queue, Round Robin is considered fair in terms of CPU allocation.

B. Race Conditions and Critical Sections

When multiple processes or threads share data, concurrent access can cause race conditions if updates are not properly coordinated. A race condition occurs when the final result depends on the timing and interleaving of operations [1]. To avoid such errors, critical sections—regions of code that access shared variables or resources—are identified, and only one process at a time is allowed to execute inside a given critical section.

Operating systems and programming languages provide synchronization primitives to enforce mutual exclusion and coordination:

- Mutex locks allow exclusive access to critical sections.
- Semaphores can be used both for mutual exclusion (binary semaphores) and for controlling access to resources (counting semaphores) [2].
- Monitors encapsulate shared data and methods, providing implicit mutual exclusion.

Classic problems such as producer–consumer, readers–writers, and dining philosophers illustrate both the need for synchronization and the risks of deadlock and starvation [1], [2].

C. Scheduling and Synchronization Together

In textbooks, scheduling and synchronization are often treated as separate topics. In actual systems, they interact: synchronization primitives operate at the level of process and thread behavior, causing processes to block until conditions are satisfied (e.g., a lock becomes free or a buffer has space), while the CPU scheduler operates at the system level, deciding which ready process should use the CPU.

Standard Round Robin does not inspect what a process is about to do. It only knows whether the process is in the READY, RUNNING, or BLOCKED state. Therefore, it may allocate time to a process that is "ready" but will almost immediately transition to "blocked" due to a locked critical section. This can lead to wasted quanta: the CPU gives a time slice to a process that cannot make progress, while delaying another process that could.

III. First Attempt: History-Based Lock-Aware Round Robin

A. Design Based on Historical Metrics

One natural extension of Round Robin is to incorporate a notion of *history*: how much time each process has spent waiting for locks and how much time each process has spent holding locks. The intuitive goal is to

- reward processes that have been blocked extensively on locks (lock victims), and
- slightly penalize processes that tend to hold locks for long periods.

Conceptually, for each process P_i, the following quantities can be tracked:

- age$[i]$: time spent in the ready queue since the process was last scheduled,
- blockTime$[i]$: total time spent blocked on locks, and
- lockHoldTime$[i]$: total time spent holding locks.

A dynamic "effective priority" can then be defined as

$$\begin{aligned}\mathbf{effectivePriority}[i] = \mathbf{basePriority}[i] &+ \boldsymbol{\alpha}\,\mathbf{age}[i] \\ &+ \boldsymbol{\beta}\,\mathbf{blockTime}[i] \\ &- \boldsymbol{\gamma}\,\mathbf{lockHoldTime}[i]\end{aligned} \tag{1}$$

where $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, and $\boldsymbol{\gamma}$ are nonnegative weights. The scheduler chooses, among ready processes, the one with highest $\mathbf{effectivePriority}[i]$, while still giving each selected process a fixed time quantum Q.

B. Penalizing Lock Holders Can Backfire

This design has an important drawback. Consider the following situation:

- Process P_1 currently holds a lock L on a critical section.
- Processes P_4 and P_5 are blocked waiting to acquire the same lock.
- $\mathbf{lockHoldTime}[1]$ is large because P_1 has already spent significant time inside the critical section.

If $\mathbf{lockHoldTime}[1]$ is used to reduce P_1's effective priority while it is still holding the lock, the scheduler may prefer to run other ready processes instead of P_1. However, P_1 is the only process that can execute the `unlock` operation and free the lock. Delaying P_1 therefore extends the time for which P_4 and P_5 remain blocked. The mechanism intended to help lock victims ends up harming them.

This suggests that a lock-aware scheduler should avoid penalizing a process while it holds a critical resource on which others depend. It is more effective to focus on avoiding wasted quanta than on assigning penalties.

IV. Hybrid Lock-Dependency-Aware Round Robin

A. High-Level Idea

Instead of penalizing lock holders, the key design question is whether it is beneficial to schedule a process that is expected to immediately block on a lock held by another process. Running such a process gives it a turn, but that turn mostly consists of discovering that it cannot enter the critical section. At the same time, the process that holds the lock may be delayed.

The proposed scheduler therefore retains Round Robin as the base policy but introduces a simple lock-dependency check:

- Two ready structures are maintained:
  - Main Ready Queue (`ReadyMain`): processes that can make useful progress if scheduled now.
  - Deferred Ready Queue (`ReadyDeferred`): processes that are ready in principle, but are known to immediately block on a lock currently held by another process.
- When a process is holding a critical-section lock and some later processes in the queue are known to soon request that same lock, those dependent processes are temporarily moved from `ReadyMain` to `ReadyDeferred`.
- Scheduling continues from `ReadyMain` using Round Robin, giving more opportunities to the lock holder and independent processes.
- Once the lock is released, the deferred processes are moved back into `ReadyMain`.

Within `ReadyMain`, processes are still rotated in Round Robin order. The temporary deferral acts as a lightweight, dependency-based priority mechanism.

B. Pseudocode

For clarity, simplified pseudocode is shown for a single lock L.

In a simulator, the last step can be approximated by annotating each process with the sequence of actions it will perform. In a real system, it could rely on compiler or programmer hints, or on approximations based on past behavior.

```
ReadyMain     <- queue of ready processes
ReadyDeferred <- empty queue
LockHolder[L] <- null

while (there exist unfinished processes) {
    if (ReadyMain is empty and not
    ReadyDeferred) {
        move all processes from ReadyDeferred
            to ReadyMain
    }

    P <- dequeue(ReadyMain)
    run P for up to quantum Q, or until event
        occurs

    if (P calls lock(L)) {
        if (LockHolder[L] == null) {
            LockHolder[L] <- P
            // P continues inside critical
    section
        } else {
            // cannot acquire lock, P becomes
    BLOCKED
            block(P)   // not in any ready
    queue
            continue   // schedule next
    process
        }
    }

    if (P calls unlock(L)) {
        LockHolder[L] <- null
        wake one process blocked on L and
    enqueue
            it into ReadyMain

        // all processes deferred only because
     of L
        // can return to main ready queue
        move all L-dependent processes from
            ReadyDeferred to ReadyMain
    }

    if (P is still RUNNABLE and quantum
    expired) {
        enqueue(ReadyMain, P)  // standard RR
    }

    // Dependency-based deferral step:
    if (LockHolder[L] != null) {
        for each Q in ReadyMain {
            if (Q will next try to lock L) {
                move Q from ReadyMain
                    to ReadyDeferred
            }
        }
    }
}
```

Fig. 1. Hybrid lock-dependency-aware Round Robin (simplified pseudocode).

V. Illustrative Behavior and Discussion

Consider five processes $P_1, \ldots, P_5$ and a lock L protecting a critical section. Assume:

- P_1 enters the critical section early and acquires L.
- P_2 and P_3 perform CPU work and do not use L.
- P_4 and P_5 will soon attempt to acquire L.

Under plain Round Robin, the ready queue might initially be

$$\text{Ready} = [P_1, P_2, P_3, P_4, P_5].$$

The scheduler gives turns to P_1, then P_2, then P_3. When it reaches P_4 and P_5, they attempt to acquire L, find it held by P_1, and block almost immediately. Their quanta are largely wasted. Meanwhile, P_1 may be delayed in obtaining additional CPU time to finish its critical section and release L.

With the hybrid scheduler, once P_1 holds L, processes P_4 and P_5 are identified as L-dependent and moved to `ReadyDeferred`. The main queue contains only P_2 and P_3, which can make progress. When P_1 resumes and eventually releases L, P_4 and P_5 are moved back to `ReadyMain`. At that point, running them is useful, because the lock is free or can soon be acquired.

This change does not alter the semantics of locks or force early unlocks. It simply avoids giving time slices to processes at moments when they are almost guaranteed to block, and prioritizes processes that can help the system move forward.

VI. Conclusion

This paper explored how a basic CPU scheduling algorithm can be made more aware of synchronization behavior, particularly critical-section contention. Starting from standard Round Robin, a history-based lock-aware scheduler was first considered, in which processes frequently blocked on locks were rewarded and processes that held locks for long periods were penalized. Analysis showed that directly penalizing a process while it holds a lock can be counterproductive: delaying the lock holder prolongs the time other processes remain blocked.

A simpler approach based on lock dependency was then developed. The final design maintains two ready queues: a main queue for processes that can make progress, and a deferred queue for processes that would block immediately on a currently held lock. By temporarily moving lock-dependent processes out of the main ready queue and reinserting them once the lock is released, the scheduler reduces wasted quanta and allows the lock holder to finish earlier. Conceptually, this creates a hybrid between Round Robin and a lightweight priority mechanism that respects synchronization dependencies while preserving fairness among active processes.

Future work includes implementing this hybrid scheduler in a discrete-event simulator and measuring metrics such as waiting time, turnaround time, and number of context switches under various producer–consumer-style workloads. Extending the idea to multiple locks and more complex dependency patterns is another interesting direction.

Acknowledgment

The authors would like to thank Ms. Maryam Sajjad, their Operating Systems course instructor at NUST SEECS, for helpful discussions and guidance while iterating on the design of this scheduler.

References

[1] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. Wiley, 2018.

[2] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed. Pearson, 2015.

[3] W. Stallings, *Operating Systems: Internals and Design Principles*, 9th ed. Pearson, 2017.