README

Changkun Jiang (1008008344)
Wu Hung Mao (1008134207)

6 OCT 2025

Starvation Discussion

In this assignment, we only have one admission thread and one execution thread for each admission queue.Since the size of admission queue is finite, the admission thread must admit jobs into the queue once the queue is empty. Similarly, execution thread must execute job once queue is full. In these two cases, the admission/execution thread that has been waiting for a lock will guarantee to acquire the lock once the other thread is put to sleep. In other cases, the admission thread/execution thread might admit/execute the job in interleaving order, which is not starvation. Hence, in **admit_jobs**, there is no possibility of starvation.

In the implementation, jobs can only acquire resource locks in increasing order. However, We do find a scenario where starvation could happen within execute_jobs.

Consider following drawing:

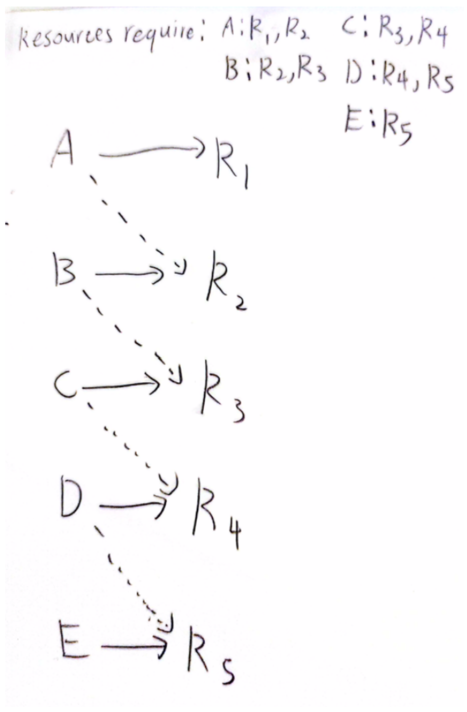

Figure 1: starvation

Left column represent jobs from different queues, right column represent resource locks. Solid line represent acquisition of a lock, and dash line represents waiting to acquire a lock.

From the image, we can see **A** waiting for **B** to release R2, which is waiting for C to release R3, ... In this case, **E**, which is holding R5, will eventually release it and break the long chain of waiting. However, it is possible that scheduling is so unfortunate that Job **F, G, H...** all require resource 5 and being chosen by scheduler to execute before D had a chance to run. Assuming there is infinite number of job to be executed, **A, B, C, D** could be starving in this specific scenario. Hence, we have come up with a solution and compare with original approach.

Jiang's solution:

Introduce another lock for the entire resource array `r_lock`

```
# when a job is ready to execute...
while True:
    if acquire r_lock:
        all_resource_acquired = True
        for l in required resources:
            if not acquire l:
                all_resource_acquired = False
                unlock all acquired resource locks
        unlock r_lock
        if all_resource_acquired:
            execute job
            unlock all acquired resource locks
```

Pros: intuitively avoids above scenario and introduce modification of resource locks array.

Cons: since ”peeking” whether a lock is acquired does not satisfy atomic operation property, it can only lock each required resource lock and keep track of them. Unlock every previous lock upon an unsuccessful attempt. In this case, these acquiring steps consumes extra steps and the bookkeeping consumes extra memory.
Requires extra `resource_array_lock[]`.

Wu's approach(our implementation):

```
# when a job is ready to execute...
for i in range(NUM_RESOURCES):
    if resource[i] is required:
        if not acquire resource_lock[i]:
            wait for signal on resource_lock[i]
execute job
unlock all acquired resource locks
```

Pros:

Guaranteed to eliminate dead locks, as all threads are acquiring resource locks in an index ascending order, it forms no circle in the waiting graph

(i.e. there will be no thread holding a ”larger” lock while waiting for a ”smaller” one)

Without a general lock to the resource lock array provides wider possibility for parallelism.

Cons: When there is infinite supply of jobs, above scenario could lead to starvation.