ACS Assignment 2

Yu Wang(ndp689) Fengmao Wang(mnp553)

December 2021

1 Techniques for Performance

1) **Batching**: Run multiple requests at once. For example, if a stage has several requests to send to the next stage, the stage can combine all of the messages into a single message and send that one message to the next stage.

   Dallying: Wait until accumulating some requests, and then run them. For example, a stage may delay a request that overwrites a disk block in the hope that a second one will come along for the same block. If a second one comes along, the stage can delete the first request and perform just the second one. (Examples are from the ACS textbook.)

   Batching may improve latency and throughput, and dallying may improve throughput when used together with batching, but typically incurs a latency penalty.

2) Concurrency may improve both throughput and latency, but must be careful with locking, correctness, which means such influence is a trade-off.

   The latency can get worse when two processes want to write on the same object and one must wait for the release of the lock.

3) Caching is an example of fast-path optimization.

   Caching provides a 'fast-path' for common requests, so it can reduce the needs for read and write operations in the memory. Then, it can reduce the latency.

   Yes, there are some trade-offs. Because, whether introducing a fast path is worth the effort is dependent on the relative difference in latency between the fast and slow path, and the frequency with which the system can use the fast path,which is dependent on the workload. If the frequency of taking the fast path is low, then introducing a fast path is likely not worth.

2 Fundamental Abstractions

1) The centralized component, a controller with an array (the size of the array is N ,and the elements in the array are address of machines), can be used to translate an address from the single address space into the address of one of the machines, like RAID (The RAID controller intercepts a request and directs the request to the corresponding disk). The name mapping can be done in constant time, because we can use the single address as an index to get the address of the machine directly in the array, and thus, it is efficient.

   When a machine breaks down, the controller can response with time-out, and if address is illegal, the controller can handle it by sending proper error messages.

2) Description of the WRITE function: first, we lookup the address of a machine in *nameArray* (name mapping) . If the address from the single address space is illegal, we just get a "Illegal" massage and it will be thrown, otherwise we will get the machine's address. Then, we write data in the machine, if an error occurs (e.g. the machine breaks down), an exception will be thrown.

```
WRITE(data, singleAddress) {
    physicalAddress = lookup(singleAddress, nameArray,
        "Illegal");
    try {
        machineWrite(data, physicalAddress);
    }
    catch(exception e) {
        throw e;
    }
}
```

   Description of the READ function: first, we lookup the address of a machine in *nameArray* (name mapping) . If the address from the single address space is illegal, we just get a "Illegal" massage and it will be thrown, otherwise we will get the machine's address. Then, we read data in the machine, if an error occurs (e.g. the machine breaks down), an exception will be thrown, otherwise we return such data.

```
READ(singleAddress) {
    physicalAddress = lookup(singleAddress, nameArray,
        "Illegal");
    try {
        content = machineRead(physicalAddress);
    }
    catch(exception e) {
        throw e;
    }
    return content;
}
```

3) Yes, the WRITE and READ operations must be atomic so that the system can be more robust. The controller should also have a lock manager, with a table of locking information, to achieve concurrency. When multiple requests are accessing the abstraction concurrently on the same object on the same machine, we use the strict 2PL rule to handle such requests.

4) a. How does this change in requirements affect your design and the faults it can tolerate? When a new machine enters, we can add its address mapping in the array, and when a machine leaves, we can remove its address mapping. But if the total number of machines exceeds N, the size of the array, the system cannot map more machines. So, the system cannot handle requests related to the exceeding part of the machines.

   b. How would you adjust the design to tolerate faults stemming from dynamic naming? We can change to use a hash map to translate an address from the single address space into the address of one of the machines instead of the array. Because the size of the hash map can be changed dynamically.

3 Serializability & Locking

1) Both of them are conflict-serializable since their precedence graphs are acyclic. We can change the initial R(X) in T2 of Schedule 1 into W(X), making a circle between T1 and T2. Thus, Schedule 1 is not conflict-serializable.

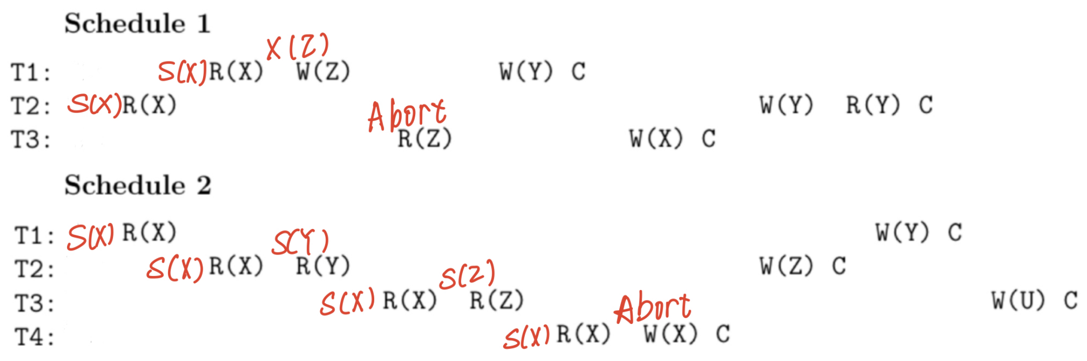
$$\begin{tikzcd}
 & T_1 \arrow[dl] \arrow[dr] & \\
T_2 \arrow[rr] & & T_3
\end{tikzcd}$$

Figure 1: Precedence graph for Schedule 1

$$\begin{tikzcd}
 & T_1 \arrow[dd] & \\
T_2 \arrow[ur] \arrow[dr] & & T_3 \arrow[ll] \arrow[dl] \\
 & T_4 &
\end{tikzcd}$$

Figure 2: Precedence graph for Schedule 2

2) The schedules cannot be generated by a scheduler using S2PL because for schedule 1, transaction 3 has a read operation on Z but there is already a X-lock on Z in T1, and for schedule 2, transaction 4 has a write operation on X but there are already S-locks on X in four transactions. (The processes are shown in Figure 3)



Schedule 1

```
                      X(Z)
T1:          S(X)R(X)   W(Z)               W(Y)  C
T2:  S(X)R(X)                                                 W(Y)   R(Y)  C
                                Abort
T3:                              R(Z)                 W(X)  C
```

Schedule 2

```
                              S(Y)
T1:  S(X) R(X)                                                      W(Y)  C
                                           S(Z)
T2:          S(X)R(X)   R(Y)                                  W(Z)  C
                                                  Abort
T3:                         S(X) R(X)   R(Z)                                      W(U)  C
T4:                                      S(X) R(X)   W(X)  C
```

Figure 3: Using S2PL

4 Optimistic Concurrency Control

1) **Scenario 1**

   T3 need to roll back.

   T1 completes before T3 starts (**Test 1**), so there is no conflict between t3 and t1.

   T2 completes before T3 begins with its write phase (**Test 2**), so we need to check WriteSet(T2) ∩ ReadSet(T3), but the intersection is $\{4\}$ not empty, so T3 need to be rolled back, and the offending object from the write set of T2 is ”4”.

2) **Scenario 2**

   T3 need to roll back.

   T1 completes before T3 begins with its write phase (**Test 2**), so we need to check WriteSet(T1) ∩ ReadSet(T3), and the intersection is empty. So, there is no conflict between t3 and t1.

   T2 completes read phase before T3 does (**Test 3**), so we need to check WriteSet(T2) ∩ ReadSet(T3) and WriteSet(T2) ∩ WriteSet(T3), but the latter is $\{3\}$ not empty, so T3 need to be rolled back, and the offending object from the write set of T2 is ”3”.

3) **Scenario 3**

   T3 will be allowed to commit.

   T1 completes before T3 begins with its write phase (**Test 2**), so we need to check WriteSet(T1) ∩ ReadSet(T3), and the intersection is empty. So, there is no conflict between t3 and t1.

   T2 completes read phase before T3 does (**Test 3**), so we need to check WriteSet(T2) ∩ ReadSet(T3) and WriteSet(T2) ∩ WriteSet(T3), and both are empty. Then, T3 will be allowed commit.