ACS Theory Assignment 1

Anders Holst, wlc376

2020-12-01

1 Question 1: Techniques for Performance

- *1. Explain the difference between concurrency and parallelism in three sentences.*

Concurrency means two or more tasks whose execution times overlap, but which does not necessarily run at the same time, eg. if the tasks are being executed on a single processor. An example is an MMU serving requests concurrently, interleaving handling of requests to hide latencies.

Parallelism simply means executing multiple tasks simultaneously on different processors/pipelines. An example of this is SIMD programming, in which multiple processors processes its own piece of a large collection of data simultaneously.

- *2. How does concurrency influence throughput?*

Concurrency can hide latencies in eg. memory requests, and in general has a positive effect on throughput. However, there are trade-offs. For example, context switches are expensive, and sometimes you waste a lot of time switching back and forth if the latency is unpredictable. There is of course also an immense overhead associated with managing concurrently executing tasks, especially if those tasks can possibly prompt other tasks to fail/restart.

- *3. Give an example of a fast path optimization and discuss its drawbacks.*

A good example is the cache hierarchies in processor architectures. The idea here is to make often used values more easily accessible. However, it has the obvious drawback of adding complexity, and this is unnecessary overhead if a program has bad (or no) cache utilization.

---

2 Question 2: Fundamental Abstractions

2.1 Name mapping scheme

- *1. Explain your name mapping scheme.*

General design

My design is largely inspired by virtual memory and paging. Assume for simplicity that K divides N. My idea is then to divide the address space evenly into K parts, which I will call *pages*, each of which has its own, unique ID in the set $\{0,1,...,K-1\}$, and each of which holds $\frac{N}{K}$ addresses of memory.

The K machines can also be labelled with ID's in the set $\{0,1,...,K-1\}$. At any given time, each active machine is responsible for one of the K pages, but machine i is not necessarily responsible for page i (eg. if page i is not in use). *In a more complex design, one machine could be made responsible for multiple pages, if that machine has enough capacity or if the cumulative size of those pages is small enough.*

A single, centralized manager, which I will call the MMU (memory management unit), handles address translation and commucation with machines.

Addresses from the single address space are first mapped to page ID's by the MMU. The MMU maintains a dynamically updated table of mappings from page ID's to machine ID's (or addresses) of machines currently maintaining those pages of memory.

Efficiency of the design

Since the single address space is partitioned in K segments of size $\frac{N}{K}$ (or $K-1$ segments of size $\frac{N}{K}$ and one segment of size $N \bmod K$ if K does not divide N), the centralized MMU would translate addresses to page ID's with a simple integer division of the address by K. This, of course, takes constant time.

The MMU would then use a (hash) lookup table to store the mappings from page ID's to machine ID's. This yields constant time translation of page ID's to machine ID's. Put together, translation of addresses from the single address space to page ID's and then to machine ID's for the machine responsible for those pages of memory takes only constant time.

Bottlenecks and tolerancy in the design

There is obviously one very big bottleneck in the design: the centralized MMU. The MMU is responsible for all client requests, name translations, and communication with machines. If the MMU fails, then everything fails.

If a single machine fails, then all pages pertaining to that machine will be unavailable until the machine reboots (if ever). In a more complex design, each of the K machine could employ a backup cache.

2.2 Pseudocode

- *2. Present API's and pseudocode for the READ and WRITE functions.*

API's

The API's are simple. For a `READ` request, all that is necessary is the read address `addr`, while for `WRITE` requests, an additional `write_val` parameter is provided.

Below is my proposed psuedocode for the API calls. I hope that the code comments speak for themselves.

```
handle_request(request_type, addr, write_val) {

  if (request_type != read || request_type != write) {
    // unrecognized request. report error to the user and exit.
  }

  else if (addr < 0 || addr >= N) {
    // client is trying to access outside the address space.
    // report this error to the user and exit.
  }

  // compute page id.
  page_id = addr / K;

  // lookup machine for this page.
  machine_id = page_to_machine_map.lookup(page_id);

  // if no machine handles this page, spawn one and associate it with this page.
  // if this is a read request, then the client will be reading uninitialized
  // memory. assume that spawn_machine() has access to some global machine ID
  // table.
  if (machine_id == null) {
    machine_id = spawn_machine();
    page_to_machine_map.add(machine_id, page_id);
  }

  // make the request. In the case that the machine (or requested
  // page on the machine) is locked, the call will simply block.
  if (request_type == read) {
    (status, read_val) = request_read(machine_id, addr);
  }
  else {
    status   = request_write(machine_id, addr, write_val);
    read_val = null;
  }

  if (status == SOME_ERROR) {
    // some error has happened in either communication,
    // or in the machine's handling of the request.
    // report this error to the user and exit.
  }

  return (status, read_val);
}

```

```
46  READ(addr) {
47    return handle_request(read, addr, null);
48  }
49
50  WRITE(addr, write_val) {
51    // since WRITE has no return value, simply ignore result of handle_request().
52    handle_request(write, addr, write_val);
53  }
```

2.3 Atomicity of READ/WRITE calls

- *Should the READ and WRITE operations be atomic?*

Before-or-after atomicity is extremely important in such a memory management system once it starts to scale to multiple clients; without it, clients' read operations might be disrupted by other clients' write operations, or segments of memory might be written simultaneously by multiple clients.

To implement before-or-after atomicity would incur considerable extra overhead in the system, especially if the system is to have the ability to recover from errors (and respect the golden rule of atomicity of never modifying the only copy of a file), since this would double the memory used, and potentially also double the total amount of available memory.

However, it might also be necessary to implement a sort of locking system on the database. I propose multi-granularity locking, since this would allow clients to claim exclusive access to entire pages, or allow multiple clients to modify single addresses of individual pages simultaneously.

2.4 Varying number of active machines

To accomodate the looser restrictions on the architecture, an implementation must be able to dynamically rename machines. With my solution, the pages that span the singular address space are statically indexed in the range $0..K-1$, but machines receive their ID only once they are spawned. A separate, internal message handler in the MMU would take care of machines despawning.

As such, I would argue that my name *does* allow machines to enter and leave.

---

3 Question 3: Serializability and locking

In the following, I will use `S(V)` and `X(V)` to denote shared and exclusive locking of a variable `V`, respectively. Since in the S2PL a commit incurs an implicit phase 2-transition and unlocking of all locks, I will omit phase 2-transitions and unlocks.

- *Draw the precedence graph for each schedule. Are the schedules conflict-serializable?*

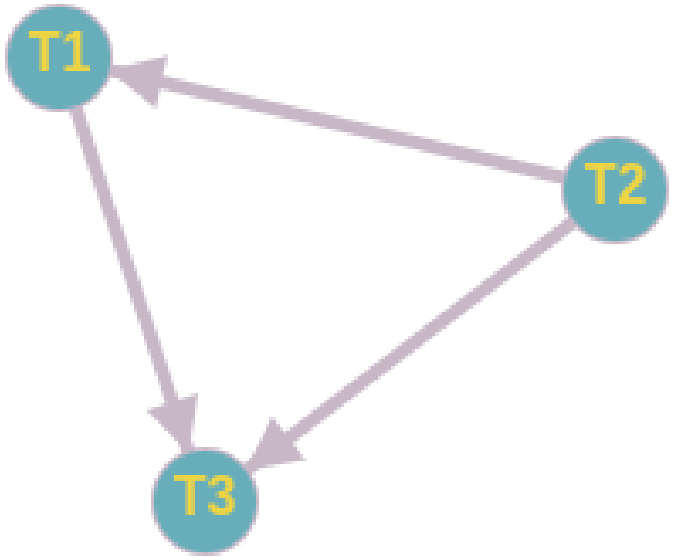
Schedule 1

```
1 T1: W(Y)            R(X) R(Z) C
2 T2:     W(Z) C
3 T3:           R(X)           R(Z) W(Y) C
```

T1 and T3 share a WR conflict on Y, producing a dependency (T1, T3).

T2 and T1 share a WR conflict on Z, producing a dependency (T2, T1).

T2 and T3 share a WR conflict on Z, producing a dependency (T2, T3).



The dependency graph has no cycles, so schedule 1 is conflict serializable.

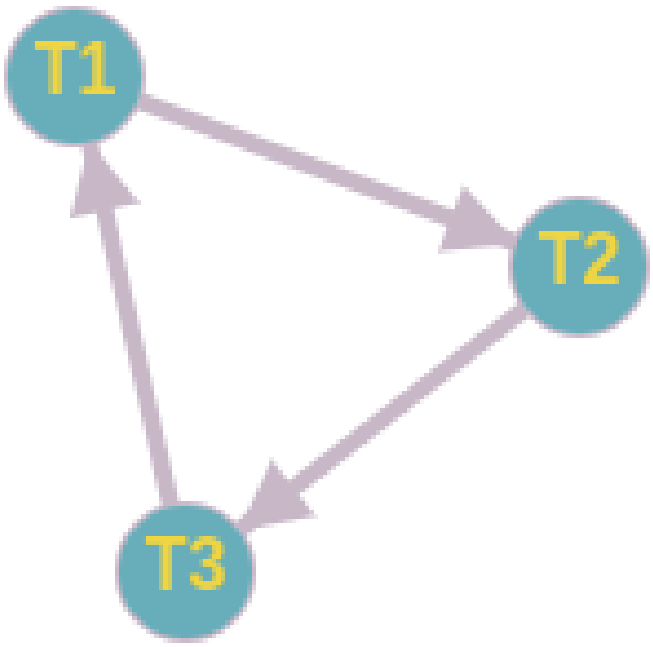
Schedule 2

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

T1 and T2 share a WR conflict on Y, producing a dependency (T1, T2).

T3 and T1 share a RW conflict on X, producing a dependency (T1, T2).

T2 and T3 share a RW conflict on Y, producing a dependency (T2, T3).



The dependency graph has a cycle in T1 -> T2 -> T3 -> T1, so schedule 2 is *not* serializable.

- *For each of the two schedules, determine whether these could have been produced by a strict two-phase locking protocol.*

Schedule 1

Yes, schedule 1 could have been produced by an S2PL protocol, for example with the locking seen in the snippet below:

```
    |time 0  |time 1 |time 2      |time 3    |time 4                |time 5
T1: |P1 X(Y) |W(Y)   |            |          |S(X) S(Z) R(X) R(Z) C |
T2: |P1      |       |X(Z) W(Z) C |          |                      |
T3: |P1      |       |            |S(X) R(X) |                      |S(Z) X(Y) R(Z) W(Y) C
```

Explanation:

time 0 T1, T2, and T3 each enter P1. T1 X-locks Y.

time 1 T1 writes Y.

time 2 T2 X-locks Z; writes Z; commits and releases X-lock of Z.

time 3 T3 S-locks X; reads X.

time 4 T1 S-locks X and Z; reads X and Z; commits and releases S-locks of X and Z.

time 5 T3 S-locks Z and X-locks Y; reads Z and writes Y; commits and releases S-lock of Z and X-lock of Y.

Schedule 2

No, schedule 2 could *not* have been produced by an S2PL protocol. As seen in the snippet below, there is an unavoidable deadlock:

```
    |time 0       |time 1    |time 2
T1: |P1 X(Z) W(Z) |          |DEADLOCK W(X) C
T2: |P1           |          |                R(Y) R(Z) C
T3: |P1           |S(X) R(X) |                           W(Y) C
```

Time-step explanation:

time 0 T1, T2, and T3 each enter P1. T1 exclusive-locks Z and writes Z.

time 1 T3 share-locks X and reads X.

time 2 T1 wants to write X and thus needs an exclusive lock of X. T3 still holds its shared lock of X and cannot release it until it has commited. Deadlock!

4 Question 4: Optimistic Concurrency Control

4.1 Notes on my answers

4.1.1 Test formulation

For each of the three scenarios, at least one of the three tests (as described in eg. the lecture 4 slides) must hold. Since in each schedule we consider three transactions, and are only concerned with validating the third transactions, I use the following specialized formulations of the validation tests:

Test 1 For $i \in \{1,2\}$, T_i must complete before T_3 starts.

Test 2 For $i \in \{1,2\}$, T_i must complete before T_3 begins its write phase **and** the intersection between WriteSet(T_i) and ReadSet(T_j) must be empty.

Test 3 For each $i \in \{1,2\}$, T_i must complete its read phase before T_3 completes its own **and** the intersection of WriteSet(T_i) and ReadSet(T_3) must be empty **and** the intersection of WriteSet(T_i) and WriteSet(T_j) must be empty.

4.1.2 Ambiguities in the given schedules

The assignment text is very ambiguous - in particular with respect to statements such as "T_i completes before T_j begins with its write phase". In this particular case, it is for example not possible to argue whether T_i finishes its read phase before T_j does.

Whenever there is such ambiguity, I will *pessimistically* assume that T_i finishes as late as possible.

4.2 Schedule 1

```
1  T1: RS(T1) = {}, WS(T1) = {3},
2    T1 completes before T3 starts.
3
4  T2: RS(T2) = {2, 3, 4, 6}, WS(T2) = {4, 5},
5    T2 completes before T3 begins with its write phase.
6
7  T3: RS(T3) = {3, 4, 6}, WS(T3) = {3}
```

T_1 completes before T_3 starts, but T_2 completes *after* T_3 starts, and so **test 1 fails**.

Test 2 fails because the intersection of WriteSet(T_1) and ReadSet(T_3) is the non-empty set {3}.

Test 3 fails for the same reason that test 2 failed (the write set of T_1 overlaps with the read set of T_3).

All three tests failed, and so T_3 must be rolled back.

4.3 Schedule 2

```
1  T1: RS(T1) = {5, 6, 7}, WS(T1) = {8},
2    T1 completes read phase before T3 does.
3
4  T2: RS(T2) = {1, 2, 3}, WS(T2) = {5},
5    T2 completes before T3 begins with its write phase.
6
7  T3: RS(T3) = {3, 4, 5, 6}, WS(T3) = {3}
```

Test 1 fails since both T_1 and T_2 end *after* T_3 starts.

T_2 completes before T_3 begins its write phase, but it is not stated whether T_1 does. I assume it does not, and thus **test 2 fails**.

T_1 completes its read phase before T_3 does, but it is *not* stated whether T_2 also does. I assume it does not, and thus **test 3 also fails**.

Again, all three tests fail, so T_3 must be rolled back.

4.4 Schedule 3

```
T1: RS(T1) = {5, 6, 7}, WS(T1) = {8},
  T1 completes read phase before T3 does.

T2: RS(T2) = {1, 2, 3}, WS(T2) = {5},
  T2 completes before T3 begins with its write phase.

T3: RS(T3) = {3, 4, 5, 6}, WS(T3) = {3}
```

Test 1 fails since both T_1 and T_2 end *after* T_3 starts.

Test 2 fails since under the specifications, T_1 does not necessarily finish before T_3 begins its write phase.

Test 3 fails since under the specifications, T_2 does not necessarily finish its read phase before T_3.

Once again, all three tests fail, so T_3 must be rolled back.