Theory Assignment - 1

**Question 2: Fundamental Abstractions**

Answer 1:

In order to organise memory with physical storage on several machines, we are going to split the top-level memory into pages. Additionally, we will maintain a map between the top-level pages and a tuple of machine identifier and local address. The mapping can be implemented to run in O(logn) time, with n being the number of allocated pages, which can be kept small by choosing suitably large page sizes.

In addition, we will for each machine maintain a list of its pages and the number of pages it has allocated. The length of the list can be used to load balance the system, by evenly splitting the memory between available machines. When a machine leaves the system, all of its pages can be copied to prevent data loss.

Answer 2:

In the pseudocode below, we assume calls to address directly into the big contiguous memory space. That is, there is no virtual memory addressing going on.

READ(addr):  
(PageNo, Offset) = (addr / page\_size, addr % page\_size) if (exists(page\_map, PageNo)):

(Machine, MachineAddr) = lookup(page\_map, PageNo)  
v = RemoteREAD(Machine, MachineAddr + Offset)  
if v == segfault: return segfault  
else if v == timeout: try again a set number of times,

if still no success, return timeout

else: return v

else:

return segfault

WRITE(addr, value):  
(PageNo, Offset) <- (addr / page\_size, addr % page\_size) if (exists(page\_map, PageNo)):

(Machine, MachineAddr) = lookup(page\_map, PageNo)

return RemoteWRITE(Machine, MachineAddr + Offset, value)

else:

if system has space for a new page:  
(Machine, MachineAddr) = allocate\_new\_page(PageNo), add(page\_map, PageNo, (Machine, MachineAddr)) RemoteWRITE(Machine, MachineAddr + Offset, value)

else:

return segfault

RemoteREAD(Machine, Addr):

SEND(Machine, \{ READ, Addr \})

RECEIVE(Machine, Value)

On timeout: return timeout

else: return Value

RemoteWRITE(Machine, Addr, Value):

SEND(Machine, \{ WRITE, Addr, Value \})

llocate\_new\_page(PageNo):  
find machine with least pages (O(logn)) try to allocate page:  
on fail:

remove machine from list of available machines for allocation

allocate\_new\_page(PageNo) on success:

return (machine, allocated page addr)

The page numbers and offsets are calculated using simple integer division and mod- ulo. The functions lookup, get and add refer to a map structure with O(logn) running times implemented with e.g a binary search tree.

**Answer 3:**

Webelievethatmemoryaccessagainsttheunifiedmemoryspaceneednotnecessarily be atomic. However, memory access to addresses in individual machines still need to preserve this basic integrity. That is, we should be able to distribute the memory access computation between machines.

**Answer 4.**

Our name mapping strategy makes an assumption about the system setup. We assume that we know the addresses, and thus also the quantity, of machines in the unified memory space. Then, we use that information to dynamically spread allocated pages over the machines.

Our system also allows for dynamic leaves and joins of machines in the memory space. Joining is simple, we just inform the system that a new machine is available with no pages allocated yet. Leaving is a bit more complicated, but can be done by iterating over the pages it had allocated. Each page should be allocated on and copied unto another machine. The leave operation should inform the system if some data could not be copied.

**Question 4: Optimistic Concurrency Control.**

**Scenario 1 :**

T1: RS(T1) = {1, 2, 3}, WS(T1) = {3},  
T1 completes before T3 starts.  
T2: RS(T2) = {2, 3, 4}, WS(T2) = {4, 5},  
T2 completes before T3 begins with its write phase. T3: RS(T3) = {3, 4, 6}, WS(T3) = {3}, allow commit or rollback?

In scenario one, we will have to rollback because of the offending object 4 in the write set of transaction two and the read set of transaction three. The conflict occurs because transaction two completes before transactions threes begins its write phase, and the intersection of their sets are non-empty.

W S(T 2) ∩ RS(T 3) = {4, 5} ∩ {3, 4, 5} (1) ={4}≠ ∅ (2)

**Scenario 2 :**

In scenario two, we will have to rollback of the offending object 3 in the write set of transaction one and the read set of transaction three. The conflict occurs because transaction one completes before transaction three begins its write phase, and the intersection of their sets are non-empty

W S(T 1) ∩ RS(T 3) = {3} ∩ {3, 4, 5, 6, 7} (3) ={3}≠ ∅ (4)

**Scenario 3 :**

T1: RS(T1) = {2, 3, 4, 5}, WS(T1) = {4},  
T1 completes before T3 begins with its write phase. T2: RS(T2) = {6, 7, 8}, WS(T2) = {6},  
T2 completes before T3 begins with its write phase. T3: RS(T3) = {2, 3, 5, 7, 8}, WS(T3) = {7, 8}, allow commit or rollback?

In scenario three, transaction three can commit as there are no offending objects.

W S(T 1) ∩ RS(T 3) = {4} ∩ {2, 3, 5, 7, 8} (5) =∅ (6) W S(T 2) ∩ RS(T 3) = {6} ∩ {2, 3, 5, 7, 8} (7) =∅ (8)