Class 18
6.852
Fall '01

Rig: [chalk, HVSB, code

~~Reading~~: ~~13.1-13.9~~ Ch. 13

Next: 17, 19

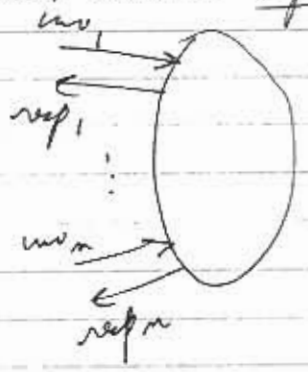
Then: 21, 22

Last Time:

Started ~~with~~ describing atomic objects

~~Defined the basic~~

like sh. vars, but admit separate invocation and response actions, on designated endpoints ("ports")



Also admit stopping failures, model as inputs on ports.

Defined the safety requirements of atomic objects 2 equivalent ways:

1. By saying that all the completed ops + some subset of the incomplete ops can get serialization points, s.t. if the inv + resp are moved to the ser pts, the result is a correct sequential exec.
2. By defining an abstract object automaton ⟶ (next page)

Today go on to describe additional, fault-tolerance properties, then describe some common algorithms for implementing atomic objects in shared memory systems.

Next time we'll talk about doing this in networks, + in general, relate shared memory + network models.

Note: atomicity (+ w-f) is a safety property.
(If fails, can be detected at some point in the execution.)
Not completely obvious — it's a complicated property.
(Proof in book uses Konig's Lemma.)

~~(Q: Alternative proof based on automaton traces?)~~

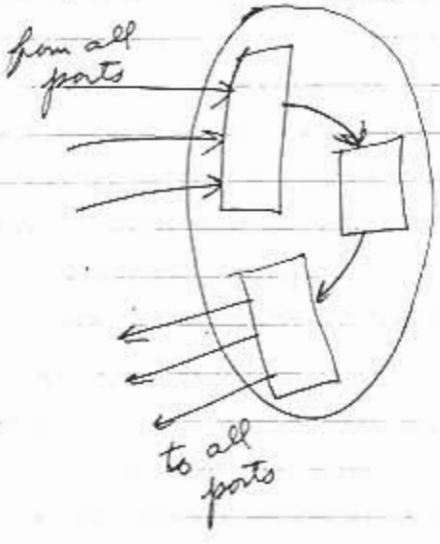
HERE

Alternative spec (for atomicity)

Equivalent in terms of the set of allowed traces
See 13.1.2

Canonical object automaton:

Keep internal copy of vble, plus delay buffers for invs + resps:



Behavior modelled by:
inv arrives, → in-buffer
perf. in separate internal step
(same as ser pt)
later returns

Has internal perform step (convenient), even though we're only interested in specifying the external behavior.

[Every trace exhibited by this aut. (with w-f Vs) is atomic, as defined above.
And every trace allowed by atomic object (with w-f Vs) is exhibited by this.]

So, this can be regarded as equivalent, alternative spec.
(Probably better than the trace-oriented one.) — but for safety only)

Liveness

Failure-free termination: (basic requirement)
In any fair ff exec of $A \times U$, every inv gets a response.
(Now fairness refers to the IOA def of fairness.)

Atomic object: w-f, atomicity, ff-term, (for all U)
And, as for consensus, can add extra fault-tolerance conds:
wait-free termination:
In any fair exec of $A \times U$, every inv on a non-failing port has a response.

Notice I didn't say anything about stops causing procs to disable actions.
I haven't even said there's a sh. mem system inside.
Could be something else, e.g. a network - & then stops might have different effects.
The point is not to specify this here - just use abstract failure actions at interface, leave considerations of what the effect of the failure is until we consider the impls.

f-failure termination: $0 \le f \le n$:
In any fair exec of $A \times U$ in which failures occur on $\le f$ ports, every inv on non-failing port gets a response.

(& book generalizes to I-failure termination, I a specific set of ports. Tolerates failures on this particular set of ports (or any subset))

Cute example of n-f ~~tg~~ atomic object

Read/increment on all ports

n-process shared memory system, vbles $x(i)$, $1 \le i \le n$ R/W vbles

N, init. 0

writeable by i, readable by all

increment$_i$: Increment local vble

Can do this using write only, by remembering the previous value written.

read$_i$: Read all the sh vars, 1 at a time, in any order. Return the sum.

wait-free ✓

atomic: Ser. pt for increment: at actual write.

Ser. pt for complete read?

Must be somewhere between inv. + resp.

Returns value $\begin{cases} \ge \text{sum at beg.} \\ \le \text{sum at end} \end{cases}$

So since increases by 1 each time, $\exists$ someplace where the actual sum is the value returned.

Put ser. pt. there.

(Of course, depends heavily on restricted operations)

Atomic objects vs sh. vbles

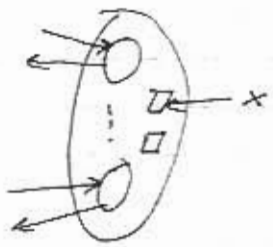
AOs aren't shared variables.

But an important basic result says it's possible to substitute them for shared vbles in a sh mem system, + the resulting system still "behaves the same."

This enables modular construction ...

The substitution: Given A, sh mem system and given atomic object B_x for each x,

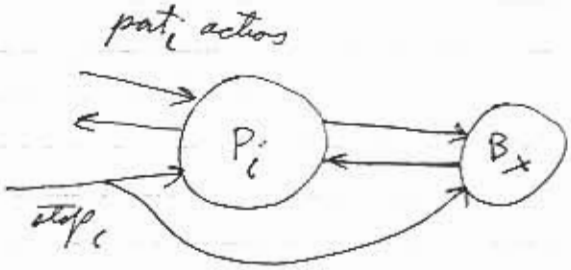
(same type, interface corresponding to the connections - who accesses what)



Trans (A) is composition of IOAs: $\begin{cases} 1 \text{ per process } i \\ 1 \text{ per sh var } x \end{cases}$

For the vbles, use the B_x's.

For the processes, use P_is, where:



← inputs of P_i:

Inputs of A on port i

Responses of all B_x's on port i

$Stop_i$

outputs of P_i:

Outputs of A on port i

Invocations to B_x on port$_i$

Steps of P_i: Simulate steps of i in A directly, except:

1. When i in A performs access to x, P_i instead issues invocation to B_x, then blocks.
   When response arrives, P_i resumes simulating i
2. When $stop_i$ occurs, all tasks of P_i disabled
   ($stop_i$ also input to the B_x's)

What is preserved by this transformation?

Theorem: Given exec α of $Trans(A) \times U$ (U = corps of all users)
$\exists$ exec α' of $A \times U$ (the sh mem system)

s.t. (a) α & α' indist. to U ($\alpha | U = \alpha' | U$)
(b) $\exists$ a each i, a $stop_i$ occurs in α iff a $stop_i$ occurs in α'.

That is, looks same to users, & also same procs fail.
(Technicality needed: at any point, either the system or the user is enabled to do something, but never both.)

Pf: Given α, try to construct corresponding α':
Insert ser. pts for all completed ops on the B_xs and some of the incomplete ops, as ensured by the atomicity defs.
& obtain the responses for incomplete ops.
Next, commute the inv & resp actions until they appear just surrounding their ser pts.
OK as far as the B_xs are concerned

$\exists$ a the P_is, might have to move actions past other actions?
(would be a problem.)
Claim not: Because P_i is blocking
& Because no inputs will arrive at P_i from U
(since it's the system's turn to take steps)

Result is still an exec of $Trans(A) \times U$. Now remove the (inv - ser pt - resp) triples & replace with single access

$\boxtimes$

Construction also preserves some liveness:

Want: α fair $\Longrightarrow$ α' fair (so liveness guarantees of α' carry over to α)

So this, need extra condition:

~~Every i for which $stop_i$ occurs in α~~

There is some set I such that

Every i for which $stop_i$ appears in α is in I

Every B_x guarantees I-failure termination

(e.g., at most f failures, & B_xs guarantee f-failure termination)

("The failures that happen are tolerated by the objects.")

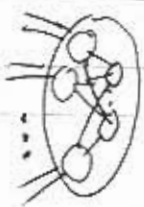
Means this construction introduces no blocking of processes.

Only possibility would be if objects don't respond

So, insist the objects always respond, except to failed processes

In special case where the objects B_x are wait-free, they never introduce blocking.

Applications of these results



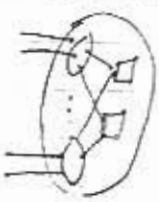
1. Special case where A itself is also an atomic object

Say with f-failure termination.

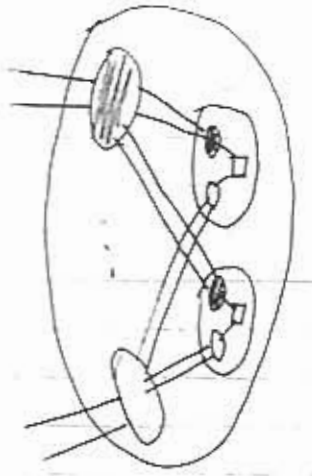
Also suppose the B_xs give f-failure termination.

Then ~~the~~ Trans (A) is also an atomic object with f-failure term.

2. Can build sh mem systems hierarchically



If B_x's are themselves sh mem systems:



Collapse the P_is and their agent processes inside the objects - get a sh mem system

In particular, can build fault-tolerant atomic objects hierarchically (transitive)

<u>Sufficient cond for atomicity</u>: Enough to prove for complete ops (implies for others)

LTTR §3.1.5

<u>RMW Atomic Objects in terms of RW sh vars</u>

<u>Non-fault-tolerant impl</u>:

Use R/W var sh. memory mutual exclusion alg.
access simulated RMW variable within critical region, which is implemented using fair R/W solution to m.e., e.g. Peterson

use separate read + write steps

<u>1-failure termination</u>:

<u>Theorem</u>: $\nexists$ sh. mem system using R/W sh vars that implements a general (function as inputs) R/M/W atomic object + guarantees <u>1-failure termination</u>.

Pf: Suppose there is, B.
Let A be RMW agreement algorithm that uses 1 shared vble, wait-free, hence 1-failure.
Get Trans(A), using B to implement the sh vars.
Solves agreement in R/W model, with 1-failure termin, contrad.
(Collapse procs so the result is formally in the sh mem model)

$\boxtimes$

Now some example algs to implement particular kinds of atomic objects in the sh mem model.

Snapshot objects
R/W objects

Atomic Snapshot objects.

Think of R/W model where each process has 1 vble it writes, others read it.
Would be nice to add this ~~to the~~ model the capability for one process to read all the vars atomically.
Atomic snapshot object provides this kind of capability.
We'll define + show how it can be implemented in terms of simple R/W shared vars.

Variable type: (for snapshot object)
values: Vector v of fixed length m, with values in some lower-level domain W.
init value: $(w_0, \ldots, w_0)$, where w_0 is init value for W

Invocations + responses

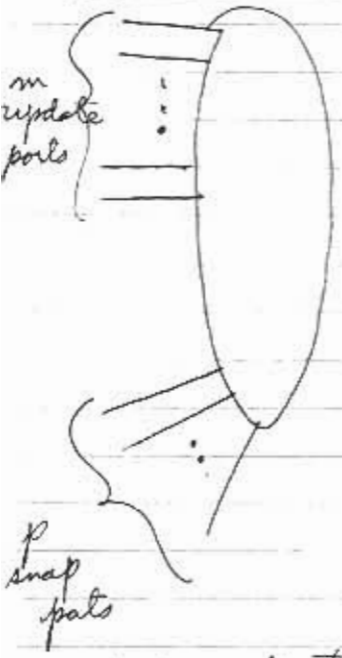
update (i, w) means write w into component i

response ack

snap

with a response that is a vector (instantaneous value)

External interface:



Assume each update port only accepts updates of corresponding vector component.

update $(i, w)_i$

$n = m + p$ total ports

Implement this atomic snapshot object using 1 process per port, 1 W m R R/W sh var.

Unbdd variable alg

$x(i)$, sh. var for update process i, $1 \le i \le m$

i can write, all can read (other updaters + snappers)

Each $x(i)$ holds: $\begin{cases} \text{element of } W \\ \text{tag (unbdd integer)} \\ \text{some other stuff} \end{cases}$

The procs are supposed to make these separate R/W vars look like a single atomic snapshot object.

First idea: To write w to component i, process i writes its own x(i). Also adds a tag that uniquely identifies the update (sequence #s, starting with 1).

Q: What can a snapper do?

Scan: read all, one at a time.

Then reads all again, + sees no changes (can tell by tags)

In this case, can return these values. The vector actually appeared in memory at some point. This will be the serialization point for the snap.

So, try using an alg. that has the snapper reading everything repeatedly, until it sees 2 identical vectors.

This is correct, if it completes.

But: the snap might never return, because of concurrent updates

However, 2nd idea: Suppose it ever sees the same x(i) having 4 different tags t_1, t_2, t_3, t_4

Then it knows that the update that wrote t_3 is entirely contained in the interval of the snap

(Because: Remnants of t_1 imply that t_2 couldn't have completed by beginning of snap, so t_3 couldn't have started.

Likewise, t_3 must finish before end of snap because t_4 begins.)

So, modify the alg so that each update process i does more:
Before it writes to $x(i)$, first executes its own embedded-snap subroutine, just like snap.
When writes (value, tag) in $x(i)$, also writes result of its embedded-snap ("view").

Now, a snap that sees four different tags for the same i, as above, can return the value of the embedded-snap associated with t_3.

Summarizing: $x(i)$ has
- val $\in W$, init w_0
- tag $\in N$, init 0
- view, vector indexed by $\{1, \ldots, m\}$ of W, init $\equiv w_0$

snap: Repeatedly reads all, any order until one of:
(1) 2 sets return same $x(i)$.tag for all i
Then snap returns the common vector of values.
(2) For some i, 4 distinct values of $x(i)$.tag are seen
Then return $x(i)$.view from the third tag.

update $(w)_i$: Perform embedded-snap, (same as snap but answer not returned to user)

Write to $x(i)$:
1. val := w
2. tag := next unused (local)
3. view := vector returned by embedded snap
+ then send ack to user

Thm: Snapshot atomic object, wait-free.

Pf: Well-formedness clear.
Wait-free termination easy - always returns by one case or the other.

Atomicity: $\exists$ ix α, exec of alg + users.
Assume WLOG all ops complete. (suffices, by lemma)
Insert ser. pts: $\begin{cases} \text{update: at point of writes} \\ \text{snap: more complicated:} \end{cases}$

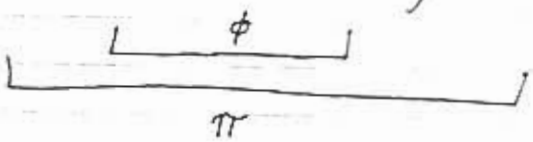
Do for all embedded-snaps as well as regular snaps:

1. Those that terminate by finding two = sets of reads
Choose any point between end of first set + beginning of second set.

2. Those that terminate by finding 4 ≠ tags
Insert ser pts for these 1 by 1, in order of their response events.

for embedded-snaps, the ~~end~~ completion of the snap

Recursively.
$\exists$ a each, say π, note that the vector returned comes from embedded-snap ϕ whose interval is totally contained within interval of π:



So by induction, ϕ has already been assigned a ser. pt.
Insert ser pt for π right after that for ϕ.

all ser pts in required intervals:

Obvious for updates.

" " first type of snap.

For second type, argue by induction on # of response events.
(containment)

Result of shrinking is a serial trace, because each snap returns the "correct" vector at its serialization point.
result of all writes up to that point

(Obvious for type 1 snaps; for type 2, follows by induction on response events.)

Complexity:

Unbdd size vars

m vars for length m vector

Time for snap: $\leq (3m+1)\,m$ sh mem accesses

$O(m^2 l)$ time

Since vbles are large, published analyses charge based on length of vector, so gives addl factor of m.
in view component

Time for update: Also $O(m^2 l)$, because of embedded-snap.

Algorithm using bdd vars:

Can do bad tag version, but tricky handshake protocol needed.

Attiya et al., LTTR (in book)

Other snapshot algs exist, with better complexity, but complicated.