Reading: 10.9-11 - Stein

Next: 13

I was covering m.e. algo - did R/W mem results (algorithms, properties, ways of analyzing lower bd.

Then Gav covered logical tree, in other networks

Now we’ll go back to sh. mem model.

Finish up some remaining details about m.e.

Quick look at sh. mem. resource alloc algo.

Then most of the hour will be spent on a major result about fault-tolerant sh. mem. computation - impossibility of consensus. (might not finish, but that’s ok)
M. E. with RMW variables - a quick overview

Results so far are very dependent on R/W data types.

But what happens if we have more powerful data types?

Results change.

E.g., R-M-W shared variables which allow 

use who value + process state to determine new value + state.

An extreme case of various more powerful primitives like

compare-and-swap, which appear in various multiprocessor memories.

Does the extremely powerful R-M-W trivialize the problem?

In a way - can get m. o. plus progress using 1 binary sh. var.

Keep testing till find \( x = 0 \); when see this set \( x := 1 \) + \( \rightarrow C \)

Exit: just do \( x := 0 \).

(Binary semaphore).

But no fairness here.

Perhaps surprisingly, to get fairness (e.g., no lockout),

problem becomes nontrivial again.

Getting fair access to semaphore does not automatically give fair access to higher-level critical sections.

E.g., suppose only one sh. var. (here, can combine WLOG)

consider how many distinct values it would need to take on

turns out that lockout freedom requires \( \Theta (n) \) values

upper + lower bound.

Essentially, need to be able to keep 1 process under ( + a constant amount of the stuff ) in shared memory, or a number of should be
A little more for intuition.

Consider a bounded bypass fairness condition, where there is some bound \( k \) on the number of times a process can go \( \rightarrow C \) while another is sitting in \( T \) (having already accessed the shared var).

Lower \( k \) on the number of values \( m \) in \( T \).

**Proof**: Suppose not, add bypass alg. \( A \) with \( \leq m - 1 \) values.

Then construct an execution:

```
1
```

Let \( p_1 \) run from initial system state, until it \( \rightarrow C \).

Then run \( p_2 \) as it enters \( T \), takes one step accessing the var.

Get \( l_3 \) by running \( p_3 \), \( \ldots \), \( l_m \) by running \( p_m \).

Since there are \( \leq m \) values of the variable, then there are two execs \( l_i \geq l_j \), \( i \leq j \) where both end in some value.

Now, since

```
l_i
```

It occurs to me \( l_1 \) to \( \ldots \), \( l_i \) only

then, \( p_1 \) reenters \( \rightarrow C \) some process \( \rightarrow C \) inf. may times inf. may times.

It happens because progress is required in fair case.

Same steps can be applied after \( l_i \) again roughly exec in which \( p_1 \rightarrow C \) inf. may times. But this isn't a fair case.
However, fences isn't required for violation of bold LTTM's property - so still get control.

0(m) alg

Basic idea then LTTM:
Divide T into buffer + main (each nonempty)
When proc $\to T$, gets into buffer.

Else proc $\to T$, Proc in main $\to C$ one at a time, any order.
When main empty, all proc in buffer $\to main$, thus emptying buffer.

How to manage all this?
Convenient to imagine dedicated supervisor (can actually simulate this with regular proc).
Supervisor keeps counts of # of proc in main + buffer.
Sh. variable has count component, it eating proc increments.
Later, supervisor "absorbs" this into its buffer count variable, to 0.

So the supervisor has enough info to decide when

- main should go main $\to C$
- buffer $\to main$

Sells proc by putting special msg (Tokes) in sh. var, which arbitrary target proc picks up.

Optimization: Can avoid having separate component in sh. var by using a priority scheme, where count takes priority over msg component for LTTR. Get $n+k$ instead of $m+k$ value.
Elmwoodly supervision - guy in C.S. takes over the role, passes it on to next guy in C.S.

Must communicate all its state info when it leaves, does so in exit protocol.

Uses ACE Riley maps for communication.

Lockout - freedom

Weaker requirement.

Can actually do in only \( \frac{n}{2} + k \) values. \( \Omega(\sqrt{n}) \) a gap.
Resource Allocation

M.O. can be viewed as problem of allocating a single non-shareable resource.

Can generalize to more resources, some sharing.

Exclusion spec \( E \): for a given set of users, is any collection of sets of users, closed under superset.

Ex: m.o.

\[ E = \{ E \subseteq \{1, \ldots, n\} : |E| \geq 1^2 \} \]

Ex: \( k \)-exclusion

\[ E = \{ E \subseteq \ldots : |E| \geq k^2 \} \]

Ex: \( V_1 \xrightarrow{} V_2 \]

\[ E = \text{all the pairs connected by edges, } + \text{ all larger sets containing such pairs} \]

\( (V_1 + V_4) \text{ do not conflict} \)

Ex: Dining Philosophers

Neighbors exclude (but no others)

(\( \neq \) set containing others)

Many exclusion specs are conveniently describable in terms of concrete resource problems:

Ex: D.P. usually formulated in terms of forks placed between adjacent philosophers.

Each philosopher needs both his adjacent forks to eat.

Since only one can have a particular fork at a time,

This leads to the condition that adjacent philosophers exclude each other.
Ex: 3rd ex. can be expressed by:

- \( V_1 \) needs resources \( A + B \)
- \( V_2 \) needs \( B + D \)
- \( V_3 \) needs \( A + D \)
- \( V_4 \) needs \( C + D \) (as \( V_1 \) and \( V_4 \) don't exclude but other pairs do)

Ex: But k-exclusion isn't expressible in this way.
For this we need to extend the notion of a resource space.
E.g. consider 2-exclusion.

\[ V_1, V_2, V_3 \] any pair or but not all 3.

Could make concrete in terms of resources by using \( R_1, R_2, R_3 \):

- \( V_1 \) needs \( R_1 \) or \( R_2 \) or \( R_3 \) (or \( R_1 \) and \( R_2 \) or \( R_2 \) and \( R_3 \) or \( R_1 \) and \( R_3 \))
- \( V_2 \) needs \( R_2 \)
- \( V_3 \) needs \( R_3 \)

With ands and ors, can express all exclusion problems concretely in terms of resources.

Anyway, given an exclusion cond. \( C \), can define an exclusion problem similar to m.e.

Same architecture, regions.

Well-formedness, as before.

Exclusion: No reachable state in which the set of users in \( C \) forms a set in \( E \).

Progress: As before: In a fair race,
1. If at least one user is in \( T \) and no one is in \( C \) then eventually someone \( \rightarrow C \).
2. If \( \rightarrow E \rightarrow R \).
No lockout - as before.

Recall this says, assuming every user in C eventually returns the resource... if i \notin T then eventually i \in C.

But now we would like to say more to capture the concurrency possibilities.

If we don't, then any m.s. alg. also solves the extended problem.

2 ways to strengthen

1. independent progress (succeed if no conflicts)

Book gives a version of this for concrete resource specs.

Basically, if i \in T and all others with conflicting resource requests remain in R, then eventually i \rightarrow C.

(similar for \bar{E})

For more general \bar{E}? An attempt:

Consider \bar{E}, the collection of all sets of processes that can coexist in C.

Inclusion spec. downward-closed.

Now say:

Suppose at some point i \in T.

And the set of all procs ever not in R, from that point on (including i) is a set in \bar{E}.

Then eventually i \rightarrow C

(Note I didn't assume everyone eventually returns the resource. So even if they stay in C, I must succeed.)
2. Semiclassical

Can get better worst-case time bounds for \( \rightarrow T \) to \( \rightarrow C \), even in the presence of conflicts, because shouldn't have to wait for everyone; only those that actually conflict.

Ex: Dining Philosophers

Dijkstra posed the problem

didn't have a solution but

used globally shared

variables with atomic

access to several vars.

Suitable for multiprocessors on one processor, not for distributed systems.

More distributed setting: Assume the edges are on edges between pairs of adjacent nodes.

Corresponds to facts.

Use RW h. vars.

Impossibility result: If all pros identical, refer to jobs by local names \( L \) and \( R \), all the h. vars have same initial values, then we have initial symmetry.

And can't break the symmetry reliably, so can't solve the problem.

Proof similar to impos of leader election in symmetric, synchronous rings.

Now don't have rounds, but can run alg. in round-robin fashion in such a round-robin fashion.

Proof by induction on \( \frac{r}{2} \) of rounds that pros remain in same state, variables have same value, after \( r \) rounds depends on only accessing our side in each step.
By progress, someone \( \rightarrow \) \( C \), then all do, violating DP exclusion.

Ex: Consider simple "symmetric" alg where

1. All pick up \( R \) first (test until they get it)
2. Then \( "~L" \) fork (wait)

\( \) thus exclusion because need both forks.

But progress fails - all might grab \( R \), then stuck.

So, need something to break symmetry: \( \text{VIDS, initial config assumptions, ...} \)

**Example algorithm:** R-L alg (Bruns, a folklore)

**DP exclusion, progress, lockout-freedom, indep., progress, & good worst-case time (indep. of \( n \)).**

Breaks symmetry by classifying frags as \( R \) or \( L \) (at least one of each).

Here, just consider simple case \( n \) even, \( R \) \& \( L \) alternate.

**The idea:**

**Odd \#:** pick up \( R \) fork first.

**Even \#:** \( "~L" \) fork.

Each has its 91 w/ trivial queue indicating who has it or wants it. First on queue has it, second \( "~" \) is waiting.

To request fork:

Put self on queue.

When see you've gotten to front, also put self on other queue.
When are you're gotten to front of that $\rightarrow C$.

So exit:
Remove index from both queues, 1 at a time.

Correctness: Exclusion obvious because read both forks.
Indep progress also easy to see
Progress + deadlock freedom follow from true for $i \rightarrow C$, which I'll sketch. Uses simple recurrence.

Claim bound on time for process to reach $C$: $3c + 18t$

Let $T = \text{max time from when any } i \rightarrow T \text{ until } i \rightarrow C$

$S = \ldots \ldots \ldots \text{ any i obtains its first fork}$
(i.e., becomes first on its queue)

$\text{until } i \rightarrow C$

Want bound on $T$.

1. Bound $T$ in terms of $S$

Suppose $i$ enters $T$. Within $L$, performs test event $T$ to try

$\text{total time } \leq L + S$.

If not, then only has the fork when it occurs.
Since it's $i$'s first fork, it's also the sub's first fork.

Add 1 time until the sub releases the fork $\leq S + C + L$.

tie to release first fork

$\text{tie in } C$

$\text{tie in } C$
When mh releases the fork, i (automatically) obtains it.
Then within add' l S, i → C.
So in this case, total time ≤ l + S + c + l + S

= \[c + 2l + 2S\]

So, combining cases, \[T \leq C + 2L + 2S\]

2. Bound S

Suppose i has just obtained its first fork.
Within l, it discovers this; within add' l t tests 2nd fork.
If obtains fork immediately, then within add' l L → C,

\[total \leq 3L\]

Otherwise, mh has the 2nd fork, and it's also the mh's 2nd fork.
Since until mh releases 2nd fork ≤ 2L + c + 2L

\[\text{mh determines it has the fork \( \rightarrow C\)}\]

3. when mh releases fork, at most l till i discovers

\[+ \ldots \ldots \ldots \ldots i \rightarrow C\]

Total for this case ≤ 2L + 2L + c + 2L + 2L

\[= c + 8L\]

\[S \leq c + 8L\]

Combining, \[T \leq 3c + 18L\]
Extends to odd # also, + other distributions of Rs + Ls around the ring.

Also, can extend to more general graphs than ring.
Consensus
Still considering asynchronous mem.
Now introduce complication of failures: process stopping.
Consensus again.
Recall sync. network model.
Consensus solvable, even for Byz. failures.
In asynchronous model, sh. mem or network, impossible.
Impossible even if ≤1 failure (+ a large numb. m of processes)
Has implications for process commit, fault diagnosis — need to use
something beyond pure asynchronous model, e.g.

Timing
probabilities
conditional termination guarantee

Problem def: Achieve, as for ms.
Votes
V, uses
V, value set

Interaction: User submits initial value in init (V); action,
gets back decision via decide (V); action.

Slightly different from earlier treatment, where we assumed
I + O in local vars.
User assumed to only perform one init (v); in an execution stopping failures. Model with input action stop. comes from "outside".

Effect is to disable all future non-input actions of P_i.

Fairness now are those in which each process that doesn't fail (+ each user) gets inf. many turns to perform l.c. steps (as before).

(just ordinary fairness, but step makes nothing enabled.)

Problem requirements:

well-formedness: Only output 1 decision, if only if init. previously occurred.

agreement: All dec. values identical.

validity: If init. acts that occur all contain same v this that v is the only possible decision value (weak foundation)

termination:

most basic: f-f termination: In any fair FF ex., in which init. events occur on all i.f.s, a decide occurs on each i.f.

Basic problem requirements: w-f, agreement, validity, FF-termination.

Want fault-tolerance too.

Strongest we might consider:

wait-free termination: In any fair ex. in which init. events occur on all i.f.s, a decide event occurs on every i.f. on which no stop occurs.
recall wait-free doooyu in Lampot's bakery alg.
This is similar.
Say i finishes regardless of whether other frees stop or not.
Also, for the main result, we'll want to consider tolerance of limited # of failures, not nec. all.
(should be easier to achieve).

1. failure termination: 0 ≤ f ≤ m
In any fair execution in which unt events occur on all ports,
if there are stop events on at most f ports,
then a decide occurs on every non-stopping port.

1. failure termination:
As the interesting special case we'll consider.

Note: Modeling failure at external interface is a good idea,
can talk about failures in specs.
Nice for modular treatment.

Ex. van types: i Consider RIW
     Problem easy with stronger RMW