Class 16
6.852
Fall '01

(15 covered by Rjei - did Ch. 18)

bring: {chalk
HW 5a

Reading: 10.9, 11 - skim
12

Next: 13

I was covering m.e. algs - did R/W mem results { algorithms, ~~proofs~~
properties, ways of analyzing
lower bd.

Then Rjei covered logical time, in asynch 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 algs.

Then most of the hour will be spent on a major result about fault-tolerant ~~sh~~ 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 vbles, which allow procs to access sh var, use vble 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.e. + progress using 1 binary sh. var:

Keep testing till find $x = 0$; when see this set $x := 1$ + → 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 index (+ a constant amount of other stuff) in shared memory.

– or, a counter of # of processes

a little more, for intuition.

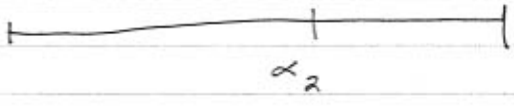
Consider a <u>bounded bypass</u> fairness condition, + progress, where there is some bound k on the number of times 1 process can go $\to C$ while another is sitting in T (having already accessed the sh. var).

<u>Lower bd on # of values n</u>

<u>Pf</u>: Suppose not, bdd bypass alg. A with $\leq n-1$ values.

Then construct an execution:

Let <u>p_1</u> run from initial system state, <u>until it $\to C$</u>.

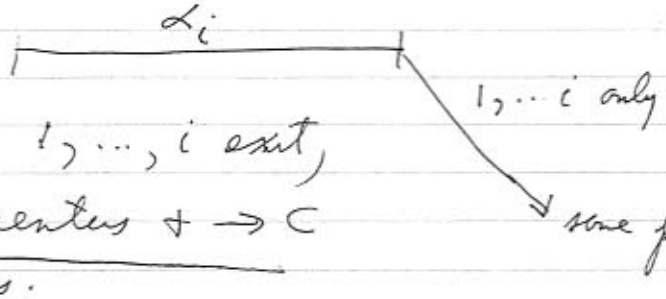


Now run <u>p_2</u> so it <u>enters T</u>, takes one step accessing the var.

Then get <u>α_3</u> by running <u>p_3</u>, ... <u>α_n</u> by running <u>p_n</u>.

Since there are $\leq n$ values of the variable, then there are <u>two execs α_i, α_j</u>, $i \leq j$, where <u>both end in same value.</u>

Now, ~~fix~~



extend α_i so $1, \ldots, i$ exit,

then <u>p_1 reenters + $\to C$ inf. many times.</u>

Happens because <u>progress is required in fair exec.</u>

<u>Same steps can be applied after α_j</u>, again yielding exec in which $p_1 \to C$ inf. many times. But <u>this isn't a fair exec.</u>

However, fairness im-T required for violation of bdd bypass property – so still get control.

☒

O(n) alg

Basic idea, then LTTR:
Divide T into buffer + main (each unordered)
When proc → T, goes into buffer.
~~at some time~~
Procs in main → C one at a time, any order.
When main empty, all procs in buffer → main, thus emptying buffer.

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

So the supervisor has enough info to decide when ~~[illegible]~~
someone should go main → C
" " " buffer → main

Tells procs by putting special msgs (tokens) in sh. var, which arbitrary ~~[illegible]~~ target process picks up.

Optimization: can avoid having separate components in sh var by using a priority scheme, where count takes priority over msg component LTTR get n+k instead of nk values

~~xx~~ <u>Elemnaly supervisor</u> – guy in C.S. takes over the role, passes it on to next guy in C.S.

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

Uses sh. rdbl mrgs for communication.

<u>Lockout-freedom</u>

Weaker requirement.

Can actually do in only $\frac{n}{2}+k$ values.

Lower bd ? $\Omega(\sqrt{n})$ } a gap.

Resource Allocation

M.e. can be viewed as problem of <u>allocating a single nonshareable resource</u>.

Can generalize to more resources, some sharing.

<u>Exclusion spec $\mathcal{E}$</u>: for a given set of users, is any collection of sets of users, closed under superset.

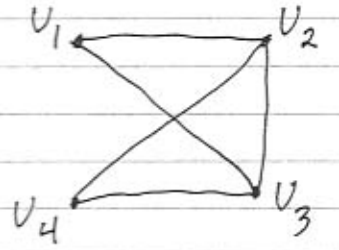
<u>Ex</u>: m.e.

$$\mathcal{E} = \{ E \subseteq \{1, \dots, n\} : |E| > 1 \}$$

<u>Ex</u>: k-exclusion

$$\mathcal{E} = \{ E \subseteq \quad \dots \quad : |E| > k \}$$

<u>Ex</u>:



$\mathcal{E}$ = all the pairs connected by edges, + all larger sets containing such pairs

(v_1 + v_4 don't conflict)

<u>Ex</u>: Dining Philosophers

Neighbors exclude (but no others)
(+ set containing nbrs)

Many exclusion specs are conveniently describable in terms of concrete <u>resource problems</u>:

<u>Ex</u>: D.P. usually formulated in terms of forks placed between adjacent philosophers.
Each philosopher needs <u>both his adjacent forks</u> 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:

U_1 needs resources A + B

U_2 " " B + D

U_3 " " A + D

U_4 " " C + D (so U_1 + U_4 don't exclude but other pairs do)

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

U_1, U_2, U_3, any pair OK but not all 3.

Could make concrete, in terms of resources, by using R_1, R_2, R_3.

U_1 needs ~~R_1 or R_2~~ $(R_1$ and $R_2)$ or $(R_1$ and $R_3)$ or $(R_2$ and $R_3)$

U_2 " ~~R_2 or R_3~~ "

U_3 " ~~R_1 or R_3~~ "

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

Anyway, given an exclusion cond. $\mathcal{C}$, 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 $\mathcal{C}$.

Progress: As before: In a fair exec,
1. If at least one user is in T and no one is in C then eventually someone → C.
2. If - - - - - - - - E ——— " "
" → R.

No lockout – as before

Recall this says, assuming every user in C eventually returns the resource ... if $i \in T$ 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.e. 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 $\boxed{p.\ 339}$

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

(+ similar for E)

For more general $\mathcal{C}$? An attempt:

Consider $\overline{\mathcal{C}}$, 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 $\overline{\mathcal{C}}$.

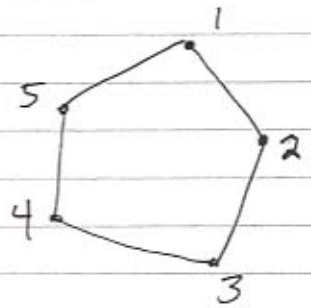
Then eventually $i \to C$

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

2. Time bound

Can get better worst-case time bds for $\to T$ to $\to C$, 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

Gave a solution, but used globally-shared vars, with atomic access to several vars.

Suitable for multiprocessing on one processor, not for dist. systems.

Mov distributed setting: Assume ~~the vars are~~ the only sh vars are on edges between pairs of adjacent nodes.

Correspond to forks.

Use RMW sh. vars.

Impossibility result: If all procs identical, refer to forks by local names L + R, + all sh 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 execution:

Prove by induction on # r of rounds that procs remain in same state, variables have same value, after r rounds

Depends on only accessing one vble in each step.

By progress, someone → C. Then all do, violating DP exclusion

Ex: Consider simple "symmetric" alg where

{ all pick up R fork first (test until they get it)
{ then " " L fork (wait)

Has exclusion because need both forks
But progress fails – all might grab R, then stuck.

So, need something to break symmetry: UIDs, initial config assumptions, ...)

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

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

Breaks symmetry by classifying procs as R or L (at least one of each).
Here, just consider simple case, n even, R + L alternate
(But can generalize.)

The idea

Odd # procs pick up R fork first.
Even # " " " L " ".

Fork has sh var with 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 see you've gotten to front of that, $\to C$.

<u>To exit:</u>

Remove index from both queues, 1 at a time.

<u>Correctness:</u> <u>Exclusion</u> obvious because need both forks.

<u>Indep progress</u> also easy to see

<u>Progress</u> + <u>lockout-freedom</u> follow from <u>time bd</u>, which I'll sketch. Uses simple recurrence.

Claim bound on time for process to reach C: $\boxed{3c + 18l}$

Let <u>T</u> = max time from when any <u>$i \to T$</u> until <u>$i \to C$</u>.

<u>S</u> = " " " " any <u>i obtains its first fork</u>

(i.e., becomes first on its queue)

until <u>$\to C$</u>.

<u>Want bound on T.</u>

<u>1. Bound T in terms of S</u>

Suppose i enters T. Within l, performs test event π to try to get first fork.

If it gets it, then within add'l time S, $i \to C$

Total time $\leq \boxed{l + S}$.

If not, then <u>nbr has the fork when π occurs.</u>

Since it's i's first fork, it's <u>also the nbr's first fork.</u>

Add'l time until the nbr releases the fork $\leq$

$$\underbrace{S}_{\text{nbr} \to C} + \underbrace{c}_{\text{time in } C} + \underbrace{l}_{\text{time to release first fork}}$$

When nbr releases the fork, i (automatically) obtains it.
Then within add'l S, $i \to C$.
So in this case, total time $\leq l + \underbrace{S + c + l} + S$

$$= \boxed{c + 2l + 2S}$$

So, combining cases, $\boxed{T \leq c + 2l + 2S}$

2. Bound S

Suppose i has just obtained its first fork.
Within l, it discovers this; within add'l l tests 2nd fork.
If obtains fork immediately, then within add'l l, $\to C$,
total $\boxed{3l}$

Otherwise, nbr has the 2nd fork, + it's also the nbr's 2nd fork.
Time until nbr releases 2nd fork $\leq 2l + c + 2l$

nbr determines it has the fork + $\to C$

time to release

From when nbr releases fork; at most l till i discovers
+ " " " " $i \to C$.
Total for this case $\leq 2l + \underbrace{2l + c + 2l} + 2l$.

$$= \boxed{c + 8l}$$

So $\boxed{S \leq c + 8l}$
Combining, $\boxed{T \leq 3c + 18l}$

Extends to odd # also, + other distributions of Rs + Ls around the ring.

Also, can extend to more general graphs than ring.

Consensus

Still considering asynch sh mem.

Now introduce complication of failures: process stopping.

Consensus again.

Recall synch network model:

Consensus solvable, even for Byz failures.

In asynch model, R/W sh. mem or networks, impossible.

Impossible even if ≤ 1 failure (+ a large number n of processes)

Has implications for process commit, fault diagnosis — need to use something beyond pure asynch model, e.g.

- timing info
- probabilities
- conditional termination guarantee

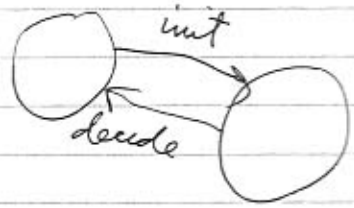
Problem def: Archit. as for m.e.

procs

V_i: users

V, value set

Interaction: User submits initial value in $init(v)_i$ action, gets back decision in $decide(v)_i$ action.



Slightly different from earlier treatment of consensus, where we assumed I + O in local vars.

User assumed to only perform one $init(v)_i$ in an execution

Stopping failures: Model with input action $stop_i$, comes from "outside".

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

Fair execs 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 stop makes nothing enabled.)

Problem requirements:

Well-formedness: Only output 1 decision, + only if $init_i$ previously occurred.

agreement: All dec. values identical.

Validity: If init acts that occur all contain same v then that v is the only possible decision value (weak formulation)

Termination:

Most basic: f-f termination: In any fair ff exec. in which init events occur on all ports, a decide occurs on each port.

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

Want fault-tolerance too.

Strongest we might consider:

wait-free termination: In any fair exec in which init events occur on all ports, a decide event occurs on every port on which no stop occurs.

Recall wait-free doorway in Lamport's bakery alg.
This is similar.
Says i finishes regardless of whether other procs stop or not.

Also, for the main result, we'll want to consider tolerance of
<u>limited # of failures</u>, not nec. all.
(should be easier to achieve).

<u>f-failure termination</u>: $0 \leq f \leq n$
In any fair execution in which inst events occur on all ports,
<u>if there are stop events on at most f ports</u>,
then a decide occurs on every non-stopping port.

<u>1-failure termination</u>:
Is the interesting special case we'll consider.

<u>Note</u>: Modeling failure at external interface is a <u>good idea</u>.
Can talk about failures in specs.
Nice for modular treatment.

<u>Sh. var types</u>: { Consider <u>R/W</u>
Problem <u>easy with stronger RMW</u>