Reading: 10.3 - 10.7
Next: 10.8, 10.9 (quick overview of 11)

(after that) 18, then back to 12013 (concurrent & atomic objects)

Last two started shared mem model

Procs interact by performing ops on shared variables
Simplicity: assume ops instantaneous
Technically that means the procs + sh sides comprise one big automaton!

Body study mutual exclusion problem in sh. mem model.

Problem: → *
Execution model: Special case of that for I/O automaton.

As usual, can express production by modeling event by another automaton (or perhaps several, 1 per part)

Commonly used variable types

read/write: Only allows access via separate read & write ops.
read-modify-write: More powerful primitive.

atomically read value in “lock”, do some computation, write something back

Get different results for different side types.

The Mutual Exclusion Problem

Allocating 1 resource among n users \( U_1, \ldots, U_n \).

( Printer, part of a DB )

\( U_i \) has 4 “regions” ( subsets of its states, often thought of syntactically in terms of portions of the code)

\[
\text{critical} \rightarrow \text{remainder} \rightarrow \text{try} \rightarrow \text{exit}
\]

Cycle: \( R \rightarrow T \rightarrow C \rightarrow E \)

Architecture:

\( V_1, V_2, \ldots, V_n \)

The \( V_i \)'s + the SM + the IOA = I/OAs.

Compare as usual.
actions at interface:

\[ \text{V.i} \quad \text{Pi} \quad \text{try} \quad \text{wait} \quad \text{wait} \quad \text{try} \quad \text{wait} \quad \text{rem} \]

Actions of \text{V.i} involve \text{Pi}.
\text{Pi} is \text{V.i}'s "agent" in the alg.

Properties to guarantee:

Assume \text{V.i} obeys the cyclic discipline. (\text{Pi} isn't the part that messes it up)
\text{e.g.} doesn't request resource when it already has it.

Then:

1. **well-formedness**
   System also obeys the cyclic discipline (safety property)

2. **mutual exclusion** (also safety)
   System never grants to two simultaneously
   (Can tell from traces)

   \( \text{In other words, in terms of states: say there's no reachable state of entire system (users \leq \text{M apps}) in which } >1 \text{ user is in C at once.} \)

3. **Progress** (fairness property)
   At any point in a fair access:
   \( \text{assume all actions of each phase in task} \)
   \begin{enumerate}
   \item If some user is in T & no user is in C then at some later point some user \( \rightarrow \text{C} \).
   \item If at least one user is in E then at some later point some user enters \( \text{R} \).
   \end{enumerate}
(Note there are additional conditions on the SMS automation system)
controls whether some user goes to C or R.

In contrast, the users determine if they go to T or E - we don't state any requirements about this.

Note we assumed that the user is fair (that all users continue to take steps).

Without this, couldn't require eventually $\rightarrow C$ or $\rightarrow R$

In general, liveliness properties require fairness hypotheses.

Contrast: W-F and M-E are safety properties (some bad event never happens)
These don't require fairness hypotheses.

More assumption: locally-controlled actions only enabled
when user is in T or E - no dedicated processes
(reasonable if few processors available, less reasonable in client-server distributed settings.
Dijkstra '65

old for operating systems

Based on Dijkstra 2-process solar

p. 265-266 code

Written here in traditional sequential pseudocode style

\[
\text{turn} \in \{1, \ldots, n\} \\
\text{flag}(i), 1 \text{ per process}, \in \{0, 1, 2\}
\]

P_sect.i's first stage

Set flag to 1, repeatedly check turn to see if turn = i

If not, or if current owner of turn is seen to be inactive,

set \text{turn} := i

otherwise

otherwise keep checking

Once sees turn = i, move to stage 2.

Stage 2

Set flag to 2, check to see no other process has flag = 2

(check in any order)

If check completes successfully, go to critical region.

When leave C

just set flag := 0

Problem with this rather manual-looking code is that it is unclear
what constitutes an atomic step.
So, I remodel to make this explicit:

p. 268-269

Actions: I + O, plus the various internal steps:
set-flag-1 is at beg
set-turn is performing test of while loop

set-flag-2 is when sets flag to 2
-check(i) is when check if 0 flag, in second stage
reset resets flag

To describe all the state changes, also need to keep p.c. info
Left implicit in usual code.
Now make explicit.

Also need set S to remember which flags have been checked
in stage 2.

So our transitions:

Try: input, so no precedent; just prepare to set flag
S-F-1: precedent just says up to this; sets flag

Test-turn: if = i, then jump ahead (don’t bother resetting)
while condition
else go to test-flag, for the guy who has turn

Test-flag: if inactive, go to set-turn (to set turn := i)
else go back to test-turn again

etc.

Look at how S is handled.
flag := 2 makes control go back to beg.
Explicit PC style looks awkward

advantages: very clear about atomicity

pc in state useful for stating invariants

Correctness

w-t: Easy

m-2: Let's argue operationally.

By contract, suppose \( U_i, U_j \) both reach \( C \)

Both must do set-flag-2 before entering \( C \)

WLOG, suppose \( i \) does it first

The flag \((i) = 2\) from that point onward.

However, \( j \) must see flag \((i) \neq 2\)

to \( \rightarrow C \)

Impossible.

progress: Tryng region argument is the interesting one.

Argue again by contradiction:

Suppose a fair exec reaches point where \{some proc. in \( T \) no one in \( C \)

\} thereafter, no one \( \rightarrow C \)

Now start removing complications:

Eventually all region changes stop

All in \( T \) have flag \( \geq 1 \)

Then everyone is in \( T + R \), all in \( T \) have flag \( \geq 1 \).
\[ a_i : \text{no region changes, all in T have flag \( \geq 1 \)} \]
\[ \text{call these "contenders" everyone in T V R} \]

Then, anytime turn is reset in \( \leq 1 \), it must be set to a contender's index.

Claim: In \( \leq 1 \), turn eventually acquires a contender's index.

\[ \square \text{Suppose not - stays non-contender forever.} \]

Consider any contender \( i \).

If ever reaches \( \leq 1 \) of while loop, \( \text{(test-turn)} \) then will set turn \( \text{:=} i \) \( \text{(sees inactive)} \)

\[ \text{why it must reach test-turn?} \]

\[ \text{If not, then succeeds in reaching C.} \]
\[ \text{But have assumed no one reaches C, contred.} \]

Once turn = contender's index, it remains = some contender's index.

May change several times \( \text{(several could be about to set)} \).

But eventually stops changing \( \text{(tests come out negative)} \), say stabilizes to \( i \)

\[ \square \text{i stabilizes at } i \]

Thereafter, all contenders \( \neq i \) mind up looping in stage \( 1 \).

\( \text{(if ever reach stage 2, it has to fail + go back to loop,} \)
\[ \text{since it can't } \rightarrow C \text{.} \]

\[ \text{But tests always fail, so stuck in the loop.} \)
Then group i has nothing left standing in the way of its \( \rightarrow C \).

Assumptive Proof of M.E

Can use invariants to prove M.E.
Must show they hold after any number of steps (finer granularity than synch modeled).

Main post: \( \exists i \mid pc_i = \text{crit } i \leq 1 \)

To prove by induction, have to strengthen:

1. If \( pc_i = \text{leave-try}, \text{crit }, \text{reset } i \) then \( 15_i = m \) (essentially in C)
2. \( \forall j, i \neq j, i \in S_j \) and \( j \in S_i \).

These 2 together easily imply M.E.

1. Is easy by induction (on # of steps in execution)

   Intersected steps, e.g.
   When check causes \( pc_i = \text{leave-try} \)

   But that \( \phi \) when \( 15_i = m \) (look at code)

2. Also provable by induction (uses some easy auxiliary lemma saying what S values go with
   what flag values + what pc values.)

   Key step: Where j gets added to \( S_j \), by check( j) event.

   Must have flag( j) \( \neq 2 \) or wouldn't add

   But then \( S_j = \phi \) (by easy invar), so \( i \notin S_j \), can't cause
   violation of invariant.
Running time

Use u.b. on time from when some process in T until someone in C.
What does "tie" mean here?
Assume u.b. of l on successive turns for process steps
(assuming set of each process in)
(same task)

\[ T \leq u.b. O(\log n) \]

? 1. In book

Follows similar argument to progress proof, but puts time lots on
everything instead of just saying "eventually".

LTTR

(Conjecture reached by supposing it doesn't finish by a
particular time & working within that time."
I'll go over another one later this shortly.

Other m.e. algorithms

Dijkstra has no fairness in granting of resource.
(Might not be critical in practice, if contention is rare.)
Also, I uses MWMR register turn
(Might want to use simpler primitive, T WMR.)

Other also address these limitations.

Peterson algo

Suite of algorithms (Guarantee lockset freedom for processes
But use MWMR rules
1. If all users always return the resource then any user that reaches T eventually enters C.
2. Any user that reaches E eventually reaches R.

Eventually: Of course, would prefer true bounds.
But this is all you can get from the general asych model.
(3 n-time bounds, would add timing assumptions, as for Dijkstra)
(+ other asych algo we've considered

Peterson 2-process alg

Cuts + clever, well-known.
Extends to n-process algorithm. (in 2 ways)

Proofs: 0 + 1 \[ T = 1 - i \]

Traditional code:
Set your own flag to 1 to announce presence
set (errant) turn := i
Wait for either flag(i) = 0 or turn \neq i

other guy not other guy grabbed turn
then

Atmospheric made explicit in prop/eff code
Also cleans up ambiguity of order of checking flag + turn (alternates).
(No over.)

Correctness:

m.e. Key is an invariant
If pc \in \{leave-tag, exit, reset\} and pc[2] \in \{check-flag, check-turn\}
then turn \neq i.

\[ T = 1 - i \]
(If i has won + i is currently a competitor then turn is set favorably for i (means set to i').

This implies m.e.
If both in C, turn has contradictory requirements.

Proof of this hint:
All steps are easy
E.g. consider successful check-turni', causing i \rightarrow \text{least-try}.
This explicitly determines that turn \neq i', as needed.
E.g. set-turni', which causes turn := i', could falsify.
But then pc; not in the assumed set.
E.g. successful check-flagi', causing i \rightarrow \text{least-try}.
But this can't happen, because flag (i') \neq 0 when i' in given region.

Progress: Easy to see:
Suppose someone in T, no one ever henceforth in C.
Then eventually stabilizes so no new region changes.
If exactly one in T, one other's flag = 0 \rightarrow \text{C}
If both in T, turn set favorably to me, \rightarrow \text{C}.

Lockout - free:
Argue that no process can \rightarrow \text{C} 3 times while the other is sitting in T (after having set flag := 1).
Then the progress condition can be used to show the one in T eventually \rightarrow \text{C}. 
Pf. Suppose i is in T having set flag := 1, remains there while i enters C 3 times.

In each of the 2nd and 3rd times, T sets turn := i but later, must set turn = i,

Means there must be ≥ 2 occurrences of set-turn i.

But set-turn i only occurs once during i's T,

Contradiction.

Complexity Analysis

Can get time bound for time from when any particular process enters T until it enters C.

\[ C + O(l) \]

Assume \( i \) is u.b. on time to hold resource (time in C)

\( 2 \) u.b. on local proc. step time

Do proof here in detail, since I left you to read the Dijkstra bound.

Pf. Suppose not.

Log \( i \) in T, doesn't \( i \) \( \rightarrow \) C for time \( \geq C + kT \), for a particular large \( k \).

(I'm being lazy here.)

Within \( 3T \), does check-flag

\[ \text{must find flag} (i) = 1 \], because otherwise \( i \rightarrow C \) within additional time \( T \), faster than assumed.

That means \( i \) is engaged in the protocol at that point, specifically:

\[ \{ \text{set-turn, check-flag, check-turn, lean, try, cut, reset} \} \]
But this means within turn has already been net to c, after reset.

3. check flag (c) = 1 (still true to p = c).

Then check flag, occurs again, within additional 0(c).

But this implies that turn stay = c and

3) Claim: at that point, have turn = c, and

So, claim: if p = c, check flag, check-turn, c, c,R3
Milestone proof, but cast as a pf-by-control in order to assume away certain branches of the "execution tree"
n-process algorithm

Can extend 2-process to n-process
2 ways (e.g.) linearly or in tournament

linear: use series of m-1 competitions, 1, 2, ..., m-1.
Unlike ordinary competition, at each competition there is just one
loser; m-k can win in k-level competition.

This means only 1 can win at level m-1, which yields m-1.

Go over hi-level code [p. 284]

Have flag indicating level you're competing at.
Have a turn for each level.

For each level:
Set flag to that level
Set that level's turn # to your index
Wait for either turn (k) ≠ your index
or no one else's flag is at this level or higher.

Detailed code, with atomically explicit: [p. 285-286]

Ambiguity resolved, in order of checking various flags + turn.
Alternates (checking all flags (arb. order)
   checking turn

at any point if sees turn favorable, goes to next level (a C)
if sees all flags <
Can generalize invariant of 2-process alg. to say:

"If process \(i\) is a winner at level \(k\)
(meaning either level \(s_1 \geq k\) or
level \(c = k + p_c \in \{\text{lose} - \text{ty}, \text{win}\}\) and some process \(j \neq i\) is a competitor at level \(k\)
(meaning either winner, or else
level \(l = k + p_c \in \{\text{check-flag, check-turn}\}\)
then turn \((k) \neq i\)."

Proof by induction as usual, as in 2-process case.

The only complication over 2-process case is that there are more steps to consider.

No longer have 1 flag, checked in 1 step
New here many flags
So must say something about what's true when a process is in the middle of checking flags
(LTTR - some help in Assertion 10.5.3.)

With this invariant, we can prove the key fact:
"For any \(k\), \(1 \leq k \leq m-1\), there are at most \(m-k\) winners for level \(k\)."
Proof: By induction, but not (as usual) on length of execution, rather, on # of levels.

Basis: \( k = 1 \)
If false for \( k = 1 \), state has all \( m \) processes as winners at level 1.

But then assertion above implies turn (1) \( \neq \) any of them, contradiction.

Inductive step: Assume for \( k > 1 \), \( 1 \leq k \leq m-2 \), show for \( k+1 \).
Suppose false for \( k+1 \), that is, strictly more than \( m-(k+1) \) are winners at level \( k+1 \).

\( \text{Call these } W \)

\[ \text{1-cap} \quad \cdots \quad \text{1-cap} \quad k+1\text{-cap} \]

Every level \( k+1 \) winner is also a level \( k \) winner.
By ind. hyp. \( \# \) of winners at level \( k \) \( \leq m-k \), so must be exactly \( m-k \).

So \( W \) = exactly the set of level \( k \) winners also.

Now, what's the value of turn \( (k+1) \)?

Assertion implies it can't be the index of any process in \( W \), (since all are winners at level \( k+1 \)).

But the value of turn \( (k+1) \) must be the index of some competitor at level \( k+1 \).

(LITR, given as assertion)

However, every competitor at level \( k+1 \) is a winner at level \( k \), so is in \( W \).

Contradiction.
Progress, Lichten-freedom

Both follow if we can show a time bound for each process, for \( T \rightarrow C \).

Claim: \( 2^{m-1} C + O(2^m m^4) \) Exponential in \( m \)

(Not sure how tight this is.)

Analysis using recurrences

\[
T(0) = \max \text{ time from when process } T \text{ tied } \rightarrow C
\]

\[
T(k) \quad 1 \leq k \leq m-1
\]

= \max \text{ time from when wins at level } k \text{ tied enters } C

Want to bound \( T(0) \)

\[
T(m-1) \leq 1
\]

Bound \( T(k) \) in terms of \( T(k+1) \)

Cases needed

Do rough sketch.

Suppose \( i \) has just won at level \( k \).

Within \( 2^m \), does set - turn \( i \rightarrow \Box \) turn \( (k+1) \) ? = i.

Case 1: turn \( (k+1) \) gets set \( i \) within "decent interval" after \( \Box \)

\[
T(k+1) + C + (2m+2) \ell
\]

then \( i \) wins at level \( k+1 \) within add. \( m \ell \)

then \( \text{within add. } T(k+1) \quad i \rightarrow C \) (use recurrence)

So \( 2 T(k+1) + C + (3m+2) \ell \) after \( \Box \) until \( i \rightarrow C \)
Case 2: The above doesn't happen.
Then (roughly speaking), no new guys enter the level \( k+1 \), and in that interval:

Any already there will find turn \( \geq \) their own index, within \( m \), so thus win at \( k+1 \).

Thus \( \rightarrow C \) within \( T(k+1) \), then reset soon after. \((c+1)\)

Thus, all these set their flags to 0 within, not too long.
And don't reach level \( k+1 \) again in this interval
(we're assumed - else they would set turn \( \neq i \)).

This allows enough remaining time in the given interval for
\( i \) to detect all the flag variables \( \leq k+1 \),
+ thus to win at level \( k+1 \).

Then \( \leq T(k+1) \) till \( C \).

Total time from \( T \) in this case:

\[
2T(k+1) + C + (2m+2) \leq
\]

Technical

Thus, in either case,

\[
T(k) \leq 2(k+1) + C + (3m+4) \leq
\]

Technicalities, e.g. extra \( 2 \leq \) to get to \( T \)

\[
T(m-1) \leq \leq
\]

Solving for \( T(0) \) gives the exponential bound.
Tournament

Proces at leaves of binary tree compete up to root.
Each process engages in log n competitions instead of n-1,
before running at root

Roughly:

Now process has flag(i) indicating level in tree.
Was I turn was turn(x) for each competition X, Boolean
for each internal node in the tree

Basically, process works at each level by:

Setting flag(i) to that level.
Setting turn to his "role" in that comp.
Left or right (0 or 1)

Wait for either:

turn (competition) = opposite role
or
all potential opponents in tree
to have flags < this level

Correctness:

m.e. similar to before

Now the key is that, at any time, at most 1 process from a
subtree rooted at level k is a winner at level k.
Time Sched: (implies no lockout, implies progress)

\[(m-1)c + O(m^2 e)\]

not bad at all.

Can proceed with another recurrence.

Basically, each level "almost" alternates, so don't get exponential now.

Bounded Bypass:

A curiosity: Even though bound on time is small, there's no bound on number of times 1 process could bypass another in \(T\).

Discuss why I show.

No contradiction.

Because of asynchrony.

Interpretation - significance of time modeling

(some processes going very fast)