Semaphores. Limits and Extensions

SE 3BB4

Ryszard Janicki

Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada

Semaphores

- Semaphore s is an integer variable that can take only non-negative values.
- The only semaphore operations are $down(s)$ ($wait(s)$, $V(s)$) and $up(s)$ ($signal(s)$, $P(s)$).

****

$down(s), wait(s), P(s)$: **if** $s > 1$ **then** $s = s - 1$
else *block execution of the calling process*
$up(s), signal(s), V(s)$: **if** *process blocked on s*
then *awaken one of them*
else $s = s + 1$

****

- Semaphores should be atomic and 'easy/fast/efficient' to implement, preferably at a low level.

Mutual Exclusion

var s: *semaphore* = 1;

P_1 : **cycle begin** P_1-instructions$_1$;
down(s);
critical region;
up(s);
P_1-instructions$_2$;
end

P_2 : **cycle begin** P_2-instructions$_1$;
down(s);
critical region;
up(s);
P_2-instructions$_2$;
end

Theory of Semaphores

- Properties of Dijkstra's semaphore operations may be characterized in the following way. Let:
  $C(s)$ - the initial value of a semaphore variable s
  $ndown(s)$ - the number of times $down(s)$ was executed
  $nup(s)$ - the number of times $up(s)$ was executed
  $npdown(s)$ - the number of times $down(s)$ was passed
- Using these notions we may define the results of actions *down* and *up* as follows:
  $down(s)$: $ndown(s) = ndown(s) + 1$:
  if $ndown(s) \leq nup(s) + C(s)$ **then** $npdown(s) = npdown(s) + 1$;
  $up(s)$: **if** $ndown(s) > nup(s) + C(s)$
  then $npdowns(s) = npdowns(s) + 1$; $nup(s) = nup(s) + 1$;

Theorem/Definition (Semaphore Invariant)

$npdown(s) = \min(ndown(s), C(s) + nup(s))$

- Anything that satisfies the equation above is a Dijkstra's semaphore!

Smokers Problem

Three smokers are sitting at a table. One of them has **tobacco**, another **cigarette paper**, and the third one has **matches** - each one has a different ingredient required to make and smoke a cigarette but he may not give any ingredient to another.
On the table in front of them, two of the three ingredients will be placed, and the smoker who has the necessary third ingredient should pick up the ingredients from the table, make a cigarette and smoke it.
Since a new set of ingredients will not be placed on the table until this action (i.e. smoking) is completed, the other smoker who cannot make and smoke a cigarette with the ingredients on the table, must not interfere with the fellow who can. Therefore, coordination is needed among smokers.

The actions of the smokers *without* coordination

X - the smoker with tobacco

α_x: pick up the paper
pick up the match
roll the cigarette
light the cigarette
smoke the cigarette
goto α_x

Y - the smoker with paper

α_y: pick up the tobacco
pick up the match
roll the cigarette
light the cigarette
smoke the cigarette
goto α_y

Z - the smoker with matches

α_z: pick up the tobacco
pick up the paper
roll the cigarette
light the cigarette
smoke the cigarette
goto α_z

'Obvious Solution' with Semaphores

Rtobacco rt: *down*(s); *up*(*paper*); *up*(*match*); *goto* rt	Rpaper rp: *down*(s); *up*(*match*); *up*(*tobacco*); *goto* rp	Rmatch rm: *down*(s); *up*(*tobacco*); *up*(*paper*); *goto* rm	← agent
Smoker with Tobacco at: *down*(*paper*); *down*(*match*); . . . *up*(*smokert*); *goto* at	**Smoker with Paper** ap: *down*(*match*); *down*(*tobacco*); . . . *up*(*smokerp*); *goto* ap	**Smoker with Matches** am: *down*(*tobacco*); *down*(*paper*); . . . *up*(*smokerm*); *goto* am	← smokers
bt: *down*(*smokert*); *up*(s); *goto* bt	bp: *down*(*smokerp*); *up*(s); *goto* bp	bm: *down*(*smokerm*); *up*(s); *goto* bm	

↑

Rule: The set of new ingredients will not be placed on the table until an appropriate action of smoking is completed

- Deadlocking sequence (not unique):
  down(*s*) in RTobacco → *up*(*s*) in RTobacco → *down*(*paper*) in Smoker with Tobacco → *up*(*matches*) in RTobacco → *down*(*matches*) in Smoker with Paper
- paper and matches have been put on the table and the process Smoker with Tobacco takes paper while the process Smoker with Paper takes matches!

Problem (Smokers' Problem)

The Smokers' Problem is to define additional semaphores and processes and to introduce appropriate down and up statements so as to make them deadlock-free. No alternation can, however, be made to the processes defining the agent. No conditional statement and assignment statement instructions may be used.

Theorem (Patil 1970)

The Smokers' Problem has no solution.

Parnas (1974) solution to Smokers' Problem!?

initially: $s = mutes = 1, t = tobacco = paper = match = smokert = smokerp = smokerm = 0, S[1] = S[2] = S[3] = S[4] = S[5] = S[6] = 0$

Rtobacco	Rpaper	Rmatch	
rt: *down*(*s*); *up*(*paper*); *up*(*match*); *goto* rt	rp: *down*(*s*); *up*(*match*); *up*(*tobacco*); *goto* rp	rm: *down*(*s*); *up*(*tobacco*); *up*(*paper*); *goto* rm	← agent
bt: *down*(*smokert*); *up*(*s*); *goto* bt	bt: *down*(*smokerp*); *up*(*s*); *goto* bp	bt: *down*(*smokerm*); *up*(*s*); *goto* bm	const- ← raints
Smoker with Tobacco at: *down*($S[6]$); $t = 0$; . . . *up*(*smokert*); *goto* at	**Smoker with Paper** ap: *down*($S[5]$); $t = 0$; . . . *up*(*smokerp*); *goto* ap	**Smoker with Matches** am: *down*($S[3]$); $t = 0$; . . . *up*(*smokerm*); *goto* am	s m ← o k e rs
dt: *down*(*tobacco*); *down*(*mutex*); $t = t + 1$; *up*($S[t]$); *up*(*mutex*); *goto* dt	dp: *down*(*paper*); *down*(*mutex*); $t = t + 2$; *up*($S[t]$); *up*(*mutex*); *goto* dp	dm: *down*(*match*); *down*(*mutex*); $t = t + 4$; *up*($S[t]$); *up*(*mutex*); *goto* dm	push- ←ers
d1: *down*($S[1]$); *goto* d1	d2: *down*($S[2]$); *goto* d2	d3: *down*($S[4]$); *goto* d3	no over- ←flow

t and $S[...]$

- t is just and integer variable, not a semaphore variable.
- The array $S[...]$ is *an array of semaphores*, a non-standard construction, very seldom implemented, however formally OK.

Who is right? Patil or Parnas

- **Both!**
- Parnas, if the *letter of law* is more important.
- Patil, if the *spirit of law* is more important.
- Unfortunately, Patil's paper was not well written and has many *implicit* assumptions that were not spelled out.
- It was implicitly assumed: no semaphore extension to arrays, no extension to many variables, no assignment statements, etc.

- The extended primitives *edown* and *eup* are atomic (indivisible) and each operates on a set of semaphore variables which must be initiated with non-negative integer values.

$edown(s_1, \ldots, s_n, \underbrace{s_{n+1}, \ldots, s_{n+m}}_{\text{inhibitor values}})$:

if for all $i, 1 \leq i \leq n, s_i > 0$ **and for all** $j, 1 \leq j \leq m, S_{n+j} = 0$
then for all $i, 1 \leq i \leq n, s_i = s_i - 1$
else block execution of calling processes.

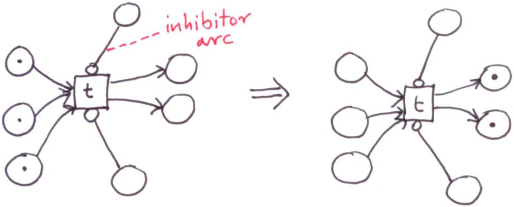
$eup(s_1, s_2, \ldots, s_n)$:
if processes blocked on $(s_1, \ldots, s_n)$
then awaken on of them
else for all $i, 1 \leq i \leq n, s_i = s_i + 1$

Theorem

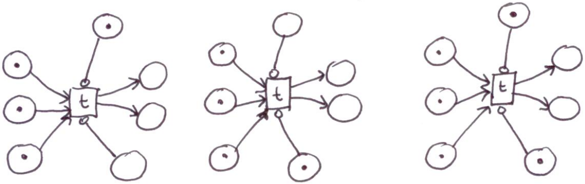
Agerwala's semaphores can simulate the action of an arbitrary Turing machine.

Inhibitor Nets

- A transition t can only be fired if all places connected by inhibitor arcs are empty.
- The transition t below can be fired as it has all input placed filled and all inhibitor places empty.



- The transition t below **cannot** be fires since its inhibitor places are not all empty (some or all contain tokens).



Nets with Inhibitor Arcs and Turing Machines

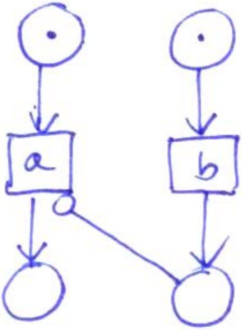
Theorem

Nets with Inhibitor Arcs are equivalent to Turing Machines.

- Agerwala's semaphores can model Nets with Inhibitor Arcs, so they can model Turing Machines as well.

'Not Later Than'

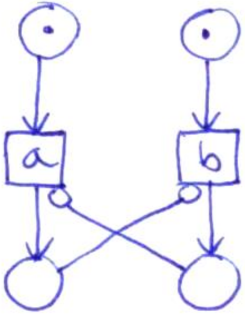
- The nets below allows the sequence $a \to b$ and the simultaneous step $\{a, b\}$, but the sequence $b \to a$ is disallowed.



- The above net models '*a is not later than b*'.

'Only Simultaneously'

- The net below allows only the simultaneous step $\{a, b\}$, neither $a \rightarrow b$ nor $b \rightarrow a$ are allowed.



- The above net models '*only simultaneous execution of a and b*'
- This net with this interpretation is a little bit controversial as the step $\{a, b\}$ does not have any sequential interpretation, so cannot be simulated by any sequential system!

Comments on Generalized Semaphores

- Both inhibitor values and $eup(s_1, s_2, \ldots, s_n)$ are rarely used in practical applications.
- Inhibitor values are needed to simulate Turing Machines.
- Releasing resources seldom needs to be done in a specific order, so $eup(s_1, s_2, \ldots, s_n)$ are not so often used.
- *Problem*: Any kind of semaphores except the classical Dijkstra's semaphores are not so easy to implement, especially on a very low level, and implementations are usually expensive and not very efficient.
- Hence, very often we only have standard Dijkstra's semaphores to use.

Smokers' Problem with Agerwala's Semaphores

Rtobacco rt: $down(s)$; $up(paper)$; $up(match)$; *goto* rt	Rpaper rp: $down(s)$; $up(match)$; $up(tobacco)$; *goto* rp	Rmatch rm: $down(s)$; $up(tobacco)$; $up(paper)$; *goto* rm	← agent
bt: $down(smokert)$; $up(s)$; *goto* bt	bt: $down(smokerp)$; $up(s)$; *goto* bp	bt: $down(smokerm)$; $up(s)$; *goto* bm	const- ← raints
Smoker with Tobacco at: $down(paper, match)$; ... $up(smokert)$; *goto* at	**Smoker with Paper** ap: $down(tobacco, match)$; ... $up(smokerp)$; *goto* ap	**Smoker with Matches** am: $down(tobacco, paper)$; ... $up(smokerm)$; *goto* am	s mo ← k e rs

- Intuition: smoker can pick only two ingredients *simultaneously*, i.e. as *atomic* one action.

Binary Semaphores. Why they are admired?

- **Simplicity, simplicity, ...**
- In FSP formalism: 'Normal' semaphores:

****

$const\ Max = M$ (must be a concrete number)
$range\ int = 0..Max$
$SEMAPHORE(N = K) = SEMA[N]$ (K is the initial value)
$SEMA[v : int] = (when(v < Max)up \rightarrow SEMA[v + 1] \mid$
$\qquad when(v > 0)down \rightarrow SEMA[v - 1])$

****

For $M = 3$ it expands to:

$SEMA[0] = (up \rightarrow SEMA[1])$
$SEMA[1] = (up \rightarrow SEMA[2] \mid down \rightarrow SEMA[0])$
$SEMA[2] = (up \rightarrow SEMA[3] \mid down \rightarrow SEMA[1])$
$SEMA[3] = (down \rightarrow SEMA[2])$

****

Binary semaphore: $M = 1$, so

$SEMA[0] = (up \rightarrow SEMA[1])$
$SEMA[1] = (down \rightarrow SEMA[0])$, we can use substitution:

****

$SEMA[0] = (up \rightarrow down \rightarrow SEMA[0])$
$SEMA[1] = (down \rightarrow up \rightarrow SEMA[1])$

****

$SEMAPHORE(N = 0) = SEMA[0]$
$SEMAPHORE(N = 1) = SEMA[1]$

Binary Semaphores. Why they are admired?

- Hence, in FSP formalism, the binary semaphores are very simple!
- If the initial value is 0 (*False*), then:
  $SEMAPHORE = (up \rightarrow down \rightarrow SEMAPHORE)$
- If the initial value is 1 (*True*), then:
  $SEMAPHORE = (down \rightarrow up \rightarrow SEMAPHORE)$
- In reality, binary semaphores are much easier to implement, especially on low level, then 'normal' semaphores.
- Usually, the implementation of binary semaphores is conceptually different, i.e. it is not just a special case of 'normal' semaphores.
- Binary semaphores are much simpler than the normal ones in virtually any high level formalism!