Lecture 3 - 4 - 5: Semaphores

© K. V. S. Prasad

TDA384/DIT391 Principles of Concurrent Programming
Chalmers Univ. and Univ. of Gothenburg

Wednesday 22 Jan 2020, Friday 24 Jan 2020 and Monday 27 Jan 2020

Invariants

"Closed circuit" Two players, Make and Prevent, move in turns on a finite rectangular grid of dots. A move links previously unlinked horizontally or vertically neighbouring dots by a dotted line (Make) or a solid line (Prevent). Make wins by making a closed circuit. Prevent wins when the grid is exhausted without a circuit. Is there a winning strategy for either?

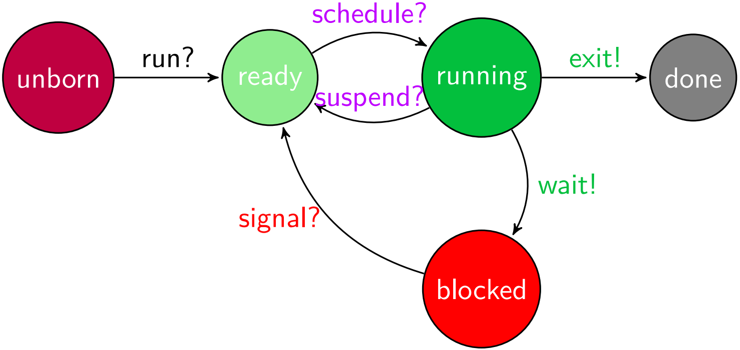
Hoare's example From "An Axiomatic Basis for Computer Programming (CACM, 1969)"

Floyd "Assigning Meanings to Programs" (Symposium on Applied Mathematics, 1967)

Turing Award Wikipedia page

Process states, showing who causes the transitions

The picture as seen by process p.



Convention, borrowed from Hoare's CSP, is ! for speech and ? for hearing.

- The run? action is by the *parent* process (who creates p).
- exit! and wait! are the only actions taken p itself.
- The signal? action is taken by a process other than p.
- schedule? and suspend? are actions taken by the invisible scheduler. No process can tell whether p is ready or running.

Definition of general semaphore

semaphore is a type or class, with atomic methods *wait* and *signal*.

Data: A pair $\langle$int V, set $L\rangle$, where V is the number of *tokens* available, (each representing a *shared resource*) and L is the set of processes *blocked* on the semaphore.
Typically, V is initialised to the total number of tokens, and L to the empty set, $\emptyset$.

Method *wait*: if $V > 0$ then V−−
else $\{L := L \cup p$; //where p is the process doing the *wait*
block $p\}$ //when p is unblocked, it completes *wait*
//by simply exiting the method.

Method *signal*: if $L = \emptyset$ then V++
else $\{L := L$-q; //where q is an arbitrary process in L
make q ready$\}$

Writers often drop L, as though the semaphore is just V. But the blocking and unblocking of processes is associated with *wait* and *signal*.

Semaphore invariants

Let semaphore S be initialised to $\langle k, \emptyset \rangle$, where $k \geq 0$. Then the following are invariant:

1. $S.V \geq 0$
2. $S.V + \#wait(S) = k + \#signal(S)$

Proof by induction on number of semaphore instructions. (Other instructions do not affect $S.V$).

1. Base: True at initialisation.
   Step: *signal(S)* can only increase $S.V$;
   wait(S) decrements it by 1 only if $S.V > 0$.
2. Base: True at initialisation; no sem actions yet.
   Step: *wait* decrements $S.V$ only if it goes through. Otherwise neither $S.V$ nor $\#wait(S)$ change.
   signal always goes through. It increments either $S.V$ or $\#wait(S)$ by unblocking a process blocked on S.

Definition of binary semaphore

A *binary semaphore* is like a general semaphore, except that V can only be 0 or 1. Method *wait* is as for general semaphore, but *signal* changes.

Data: A pair ⟨bool V, set L⟩, where V=0 (resp. 1) means the *shared resource* is (un)available.
Typically, V is initialised to 1 (available), and L to $\emptyset$.

Method *wait*: if V = 1 then V:= 0
else $\{L := L \cup p$; //where p is the process doing the *wait*
block $p\}$

Method *signal*: if V = 1 then *undefined!*
else {if $L = \emptyset$ then V:= 1
else $\{L := L$-q; //where q is an arbitrary process in L
make q ready}
}

The semaphore invariants hold for binary semaphores too.

CS problem for two processes, with semaphores

Reminder: we require that the program satisfy

- *mutex property*: if *p* is at p3 (abbr. "p3"), then $\neg$ q3
- *deadlock free*: p2 $\wedge$ q2 $\rightarrow$ p and q will not both be stuck waiting (i.e., p or q will progress to CS)
- *starvation free*: p2 $\rightarrow$ p will progress to CS

binary sem $S := \langle 1, \emptyset \rangle$	
process p	process q
while *true* {	while *true* {
p1: NCS;	q1: NCS;
p2: *wait(S)*; //entry protocol	q2: *wait(S)*; //entry protocol
p3: CS;	q3: CS;
p4: *signal(S)*; //exit protocol	q4: *signal(S)*; //exit protocol
};	}

Abbreviated CS program with binary semaphore

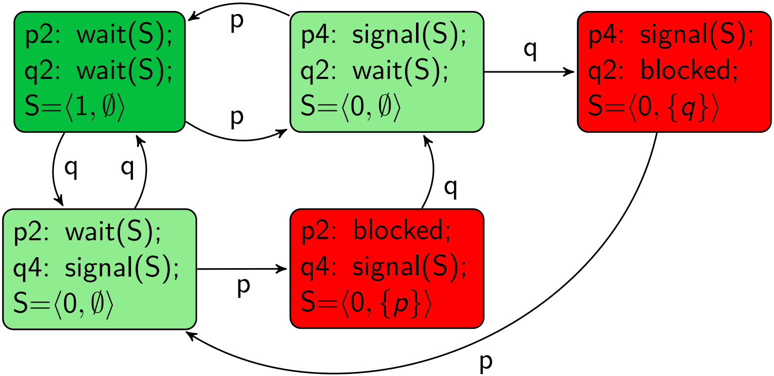
We remove the uninteresting commands to reduce the number of states we have to reason about, giving

binary sem $S := \langle 1, \emptyset \rangle$	
process p	process q
while *true* {	while *true* {
p2: *wait(S)*; //entry protocol	q2: *wait(S)*; //entry protocol
p4: *signal(S)*; //exit protocol	q4: *signal(S)*; //exit protocol
};	}

Reminder: At p4, *p* is yet to execute its exit protocol, so it is in its CS. Thus we require that the program satisfy

- *mutex property*: if *p* is at p4 (abbr. "p4"), then $\neg$ q4
- *deadlock free*: p2 $\wedge$ q2 $\rightarrow$ p and q will not both be stuck waiting (i.e., p or q will progress to CS)
- *starvation free*: p2 $\rightarrow$ p will progress to CS

State diagram, abbreviated CS program with binary sem



- The start state is at top left
- In the red states
  - one process is blocked, so only the other can move
  - only one move, by the process blocking itself, leads to a red state.
- In the green states, both can move
  - From the light green states, the system either moves back to the start state, or to a blocking state.

Correctness of semaphore CS program from state diagram

Mutex There is no state with p4 and q4. We can draw such a state, but it is not *reachable* from the start state of the program.

Deadlock There is no state where both processes are blocked. There is always a move from every reachable state.

Starvation If *p* is blocked, then *q* is poised to do a *signal*, i.e., *q* is in its CS. So it must in a finite time exit its CS (i.e., do the *signal*), and so in a finite time lead *p* into its CS.
If *p* is poised to do a *wait*, an unfair scheduler may let *q* loop around *wait* and *signal*. This is the only loop where *p* makes no progress to its CS. Since *p* is always ready to do its *wait*, a fair scheduler must let it act eventually and lead to ts CS.

Questions to ponder

Why is the semaphore defined this way? Why not let *signal* always increment *S.V* and let someone else (who?) get a waiting process to retry *wait*?

There are many other patterns to discover in the state diagram. Why 5 states? How odd that such a symmetric program produces an odd number of states!

Variant of semaphore invariant for the CS program

- Lemma: The initial value k of *S.V* is 1 in this program, so the 2nd semaphore invariant becomes $S.V + \#wait(S) = 1 + \#signal(S)$.

  Let $\#\text{CS} = \#wait(S) - \#signal(S)$; it is the number of processes in their CSs. Then $\#\text{CS} = 1 - S.V$.

  Then for this program, $\#\text{CS} + S.V = 1$ is another form of the 2nd semaphore invariant.

Correctness of semaphore CS program by invariants

- Mutex: The 1st semaphore invariant is $S.V \geq 0$, so $\#\text{CS} \leq 1$.
- Deadlock: If we are in deadlock, both processes are blocked, so it must be that $S.V{=}0$. But also $\#\text{CS} = 0$. Contradicts the above, so deadlock is not possible.
- Starvation: If p is waiting to do the *wait*, a fair scheduler must let it do that eventually.
  Then suppose p is starved, so $S.V{=}0$ and $p \in S.L$. Then because $\#\text{CS} + S.V = 1$, it follows that $\#\text{CS} = 1$ and q is in its CS, and $S.L = \{p\}$. Then q has to do a *signal(S)* and thus lead p to its CS.

Producer-consumer (PC) infinite buffer, with semaphores

An infinite buffer B holds items produced by *producer*, p, and consumed by *consumer*, c. While p can always act, c must wait if B is empty. Semaphore N is used to ensure this.

queue of int $B := \emptyset$ sem $N := \langle 0, \emptyset \rangle$	
process p	process c

int *d*;
while *true* {
p1: *append(d, B)*;
p2: *signal(N)*; //prod protocol
};
	int *d*;
	while *true* {
	c1: *wait(N)*;//cons protocol
	c2: *d:= take(B)*;
	};

Note: p does the *signal(N)*, while c does the *wait(N)*. Note also that the CS and protocols occur in different orders in p and c.

Since the buffer can grow indefinitely, the state diagram can too. So we will need meta-arguments about the diagram.

Producer-consumer (PC) infinite buffer, invariants

We begin with a simplifying assumption: make the two actions of p into one atomic action, and similarly for q. The assumption is for pedagogical reasons; it can be removed!

Then $N.V = \#B$ is an invariant. True initially. Every atomic action by p increments both $N.V$ and $\#B$. Every atomic action by c decrements both $N.V$ and $\#B$.

So *PC safety*: c never removes an item from an empty buffer.

Deadlock: Only c can block, and it won't as long as p produces. (p is allowed to stop; that is not a deadlock).

Starvation: Only c can block, and with a fair scheduler, it can always act as long as B is non-empty.

The last two arguments are degenerate cases.

Producer-consumer (PC) finite buffer, with semaphores

A finite buffer *B* holds up to *N* items produced by *producer*, *p*, and consumed by *consumer*, *c*. The conditions: *c* must wait if *B* is empty, and *p* must wait if *B* is full. Semaphores *E* and *F* are used to ensure this.

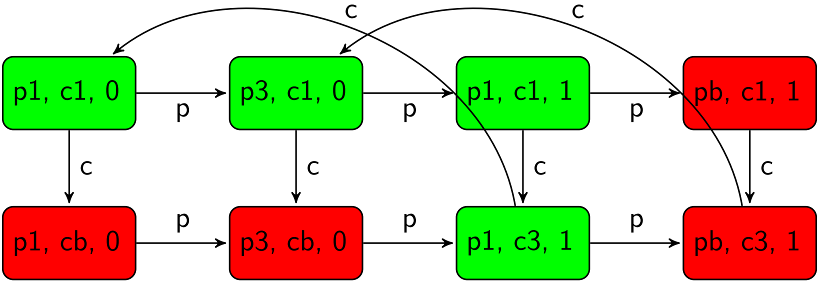
queue [capacity N] of int $B := \emptyset$ sem $E := \langle 0, \emptyset \rangle$, sem $F := \langle N, \emptyset \rangle$	
process p	process c

int *d*;
while *true* {
p1: *wait(F)*; //pre-protocol
p2: *append(d, B)*;
p3: *signal(E)*; //post-protocol
};
	int *d*;
	while *true* {
	c1: *wait(E)*; //pre-protocol
	c2: *d:= take(B)*;
	c3: *signal(F)*; //post-protocol
	};

NB: *p* does *wait(F)* and *signal(E)*, while *q* does *wait(E)* and *signal(F)*.

The *PC safety* requirement is that *c* never removes an item from an empty buffer, and that *p* never puts an item into a full buffer.

State diagram, abbreviated PC program, 1-place buffer



- *p1: wait(F)*, *p1: blocked* and *p1: signal(E)* are the three states of *p*, and similarly for *q*. The third parameter in each state notes whether there is an item in the buffer.
- The start state is at top left
- In the red states
  - one process is blocked, so only the other can move

Producer-consumer (PC) finite buffer, invariants

We begin with a simplifying assumption: make the two actions of p into one atomic action, and similarly for q. For pedagogical reasons; the assumption can be removed!

Then $N.V = \#B$ is an invariant. True initially. Every atomic action by p increments both $N.V$ and $\#B$. Every atomic action by c decrements both $N.V$ and $\#B$.

So *PC safety*: c never removes an item from an empty buffer.

Deadlock: Only c can block, and it won't as long as p produces. (p is allowed to stop; that is not a deadlock).

Starvation: Only c can block, and with a fair scheduler, it can always act as long as B is non-empty.

The last two arguments are degenerate cases.

Locks

- A *lock* is a binary semaphore
  - where only the process that does a *lock* action can do the corresponding *unlock* action.
  - there is no queue of waiting processes, so *unlock* cannot pass the lock on directly to a waiting process
  - Easy solution for the CS problem. PC problem not easy.
  - For more on locks, see Carlo/Sandro Lecture 2, frame 25 onwards.
- For a binary semaphore,
  - no thread owns it
  - consecutive P (or V) operations will be blocked
  - calls to P and V can be made by different threads
- for a lock (also called a mutex)
  - a thread that owns a lock can invoke lock operations again without being blocked
  - The owner for calls to lock and unlock must be the same thread

Monitors, protected objects

- Both monitors and Protected Objects (PO's) combine the object idea with synchronisation. Only one entry at a time.
  - What if producer enters monitor and then discovers buffer full? It waits on the *condition variable* (queue) "not-full".
  - Leads to two kinds of scheduling disciplines.
  - Messy. Deprecated.
  - For more on monitors, see Ben-Ari's slides, or Carlo/Sandro Lecture 5.
- A PO instead has a *guard* on each entry. For producer it is "not-full". Wait before entry for guard to become true.
  - Upon any exit, run-time re-checks all guards.
  - For more on PO's, see Ben-Ari's slides.

NATO Advanced Study Institute Author: Genuys, F
https://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.h