SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

CONCURRENT SYSTEMS
LECTURE 2

Prof. Daniele Gorla

Semaphores

Object: entity with an implementation (hidden) and an interface (visible), made up of a set of operations and a specification of the behaviour (usually specified in a sequential way – e.g., as a set of legal executions).

Concurrent: if the objerct can be accessed by different processes

Semaphore: is a shared counter S accessed via primitives *up* and *down* s.t.:

1. It is initialized at $s0 \geq 0$
2. *up* atomically increases S
3. *down* atomically decreases S, provided that it is greater than 0; otherwise, the invoking processes is blocked and waits.

Invariant: $S = s0 + \#(S.up) - \#(S.down)$

Main use: prevent busy waiting (suspend processes that cannot perform *down*)

- **Strong**, if uses a FIFO policy for blocking/unblocking processes, **weak** otherwise
- **Binary**, if it is at most 1 (so, also *up* are blocking)

2 underlying objects:

- A counter, initialized at s0
- A data structure (typicaly, a queue), initially empty, to store suspended proc's

Semaphores: ideal implementation

```
S.down() :=
   S.counter--
   if S.counter < 0 then
        enter into S.queue
        SUSPEND
   return
```

```
S.up() :=
   S.counter++
   if S.counter ≤ 0 then
        activate a proc from S.queue
   return
```

Remark 1:

- if S.counter ≥ 0, then this is the number of proc's that can perform down without suspending
- If S.counter < 0, then this tells us the number of proc's that are suspended in S

Remark 2: all operations are in MUTEX

Semaphores: actual implementation


```
Let t be a test&set register initialized at 0

S.down() :=
   Disable interrupts
   wait S.t.test&set() = 0
   S.counter--
   if S.counter < 0 then
        enter into S.queue
        S.t ← 0
        Enable interrupts
        SUSPEND
   else S.t ← 0
        Enable interrupts
   return

S.up() :=
   Disable interrupts
   wait S.t.test&set() = 0
   S.count++
   if S.count ≤ 0 then
        activate a proc from S.queue
   S.t ← 0
   Enable interrupts
   return
```

Remark: the interrupts are disabled only for efficiency issues (not to interrupt the semaphore operations with other – totally unrelated – operations).

(Single) Producer/Consumer

It is a shared FIFO buffer of size k. Internal representation:

- BUF[0,…,k-1] : generic registers (not even safe) accessed in MUTEX
- IN/OUT : two variables pointing to locations in BUF to (circularly) insert/remove items, both initialized at 0
- FREE/BUSY : two semaphores that count the number of free/busy cells of BUF, initialized at k and 0 respectively.

```
B.produce(v) :=
    FREE.down()
    BUF[IN] ← v
    IN ← (IN+1) mod k
    BUSY.up()
    return
```

```
B.consume() :=
    BUSY.down()
    tmp ← BUF[OUT]
    OUT ← (OUT+1) mod k
    FREE.up()
    return tmp
```

(Multiple) Producers/Consumers

Consider this solution:

We have two extra semaphores SP and SC, both initialized at 1

```
B.produce(v) :=
    SP.down()
    FREE.down()
    BUF[IN] ← v
    IN ← (IN+1) mod k
    BUSY.up()
    SP.up()
    return
```

```
B.consume() :=
    SC.down()
    BUSY.down()
    tmp ← BUF[OUT]
    OUT ← (OUT+1) mod k
    FREE.up()
    SC.up()
    return tmp
```

It is correct, but inefficient.

→ reading from/writing into the buffer can be very expensive!

→ Accessing BUF in MUTEX slows down the implementation

→ so, it would be ideal to parallelize accesses to different locations

(Multiple) Producers/Consumers

Actually, producers and consumers must mutually exclude only when they look for the cell to be written/read, respctively.

```
B.produce(v) :=
    FREE.down()
    SP.down()
    i ← IN
    IN ← (IN+1) mod k
    SP.up()
    BUF[i] ← v
    BUSY.up()
    return
```

```
B.consume() :=
    BUSY.down()
    SC.down()
    o ← OUT
    OUT ← (OUT+1) mod k
    SC.up()
    tmp ← BUF[o]
    FREE.up()
    return tmp
```

Problem: 1 PROD, 2 CONS, 2 cells

- P writes *cell0* , IN←1 , BUSY← 1 ; P writes *cell1* , IN←0 , BUSY← 2
- C1 starts reading *cell0* (but it is very slow)
- C2 reads *cell1* (quickly) and so FREE ← 1
- P thinks that it can go on writing and goes to *cell0*, that however is still busy (with C1 reading)

(Multiple) Producers/Consumers

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Assume 2 arrays (FULL / EMPTY) of booleans, initialized at ff and tt, resp

```
B.produce(v) :=
    FREE.down()
    SP.down()
    while ¬EMPTY[IN] do
          IN ← (IN+1) mod k
    i ← IN
    EMPTY[IN] ← ff
    SP.up()
    BUF[i] ← v
    FULL[i] ← tt
    BUSY.up()
    return
```

```
B.consume() :=
    BUSY.down()
    SC.down()
    while ¬FULL[OUT] do
          OUT ← (OUT+1) mod k
    o ← OUT
    FULL[OUT] ← ff
    SC.up()
    tmp ← BUF[o]
    EMPTY[o] ← tt
    FREE.up()
    return tmp
```

This solves the previous problem:

- the last write of P will discover that *cell0* is still not empty (since C1 declares it empty only when it finishes reading it)
- So, P will go to write into *cell1* (that indeed has been emptied by C2)

Monitors

Semaphores are hard to use in practice because quite low level

Monitors provide an easier definition of concurrent objects at the level of Prog. Lang.

- A concurrent object that guarantees that at most one operation invocation at a time is active inside it
- Internal inter-process synchronization is provided through *conditions*
- **Conditions** are objects that provide the following operations:
  - *wait*: the invoking process suspends, enters into the condition's queue, and releases the mutex on the monitor
  - *signal*: if no process is in the condition's queue, then nothing happens. Otherwise
    - Reactivates the first suspended process
    - suspends the signaling process that however has a priority to re-enter the monitor (w.r.t. processes that are suspended on conditions)

Implementation through semaphores

- A semaphore MUTEX init at 1 (to guarantee mutex in the monitor)
- For every condition C, a semaphore SEM_C init at 0 and an integer N_C init at 0 (to store and count the number of suspended processes on the given condition)
- A semaphore PRIO init at 0 and an integer N_{PR} init at 0 (to store and count the number of processes that have performed a signal, and so have priority to re-enter the monitor)

1. Every monitor operation starts with `MUTEX.down()` and ends with
   `if NPR > 0 then PRIO.up() else MUTEX.up()`
2. `C.wait() :=`
```
      NC++
      if NPR > 0 then PRIO.up() else MUTEX.up()
      SEMC.down()
      Nc--
      return
```
3. `C.signal() :=`
```
      if NC > 0 then  NPR++
                      SEMC.up()
                      PRIO.down()
                      NPR--
      return
```

Rendez-vous through monitors

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Rendez-vous is a concurrent object associated to *m* control points (one for every process involved), each of which can be passed when all processes are at their control points.

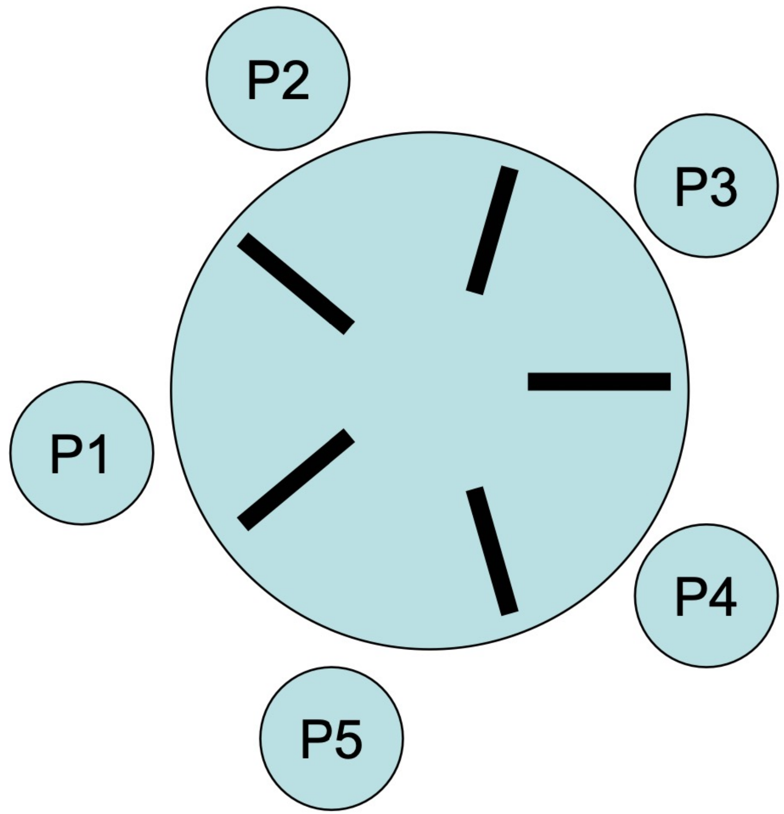
The set of all control points is called ***barrier***.

```
monitor RNDV :=
        cnt ∈ {0,…,m} init at 0

        condition B

        operation barrier() :=
                cnt++
                if cnt < m then B.wait()
                           else cnt ← 0
                B.signal()
                return
```

Dining Philosophers (Dijkstra, 1965)

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

- *N* philosophers seated around a circular table
- There is one chopstick between each pair of philosophers
- A philosopher must pick up its two nearest chopsticks in order to eat
- A philosopher must pick up first one chopstick, then the second one, not both at once



PROBLEM: Devise a deadlock-free algorithm for allocating these limited resources (chopsticks) among several processes (philosophers).

A non-deadlock-free solution

A simple algorithm for protecting access to chopsticks:

each chopstick is governed by a mutual exclusion semaphore that prevents any other philosopher from picking up the chopstick when it is already in use by another philosopher

```
semaphore chopstick[5] initialized to 1
Philosopher(i) :=
        while(1) do
                chopstick[i].down()
                chopstick[(i+1)%N].down()
                // eat
                chopstick[(i+1)%N].up()
                chopstick[i].up()
```

Guarantees that no two neighbors eat simultaneously, i.e. a chopstick can only be used by one its two neighboring philosophers

We can have deadlock if all philosophers simultaneously grab their right chopstick

Deadlock-free solutions

Break the symmetry of the system:

- All philosophers first grab their left-most chopstick, apart from one (e.g., the last one) that first tries to grab the right-most one
- odd philosophers pick first left then right, while even philosophers pick first right then left
- allow at most n-1 philosophers at the same table when there are n resources

We shall also see a solution where symmetry is not broken

- allow a philosopher to pick up chopsticks only if both are free. This requires protection of critical sections to test if both chopsticks are free before grabbing them.

→ this will be easily implemented through a monitor

Solution 1

Give a number to forks and always try with the smaller

→ all philosophers first pick left and then right, except for the last one that first picks right and then left.

```
semaphores fork[N] all initialized at 1;
Philosopher(i) :=
   Repeat
         think;
         if (i < N-1) then
                 fork[i].down();
                 fork[i+1].down();
         else
                 fork[0].down();
                 fork[N-1].down();
         eat;
         fork[(i+1)%N].up();
         fork[i].up();
```

Solution 2

Odd philosophers first pick left and then right, even philosophers first pick right and then left.

```
semaphores fork[N] all initialized at 1;
Philosopher(i) :=
   Repeat
         think;
         if (i % 2 == 0) then
                  fork[i].down();
                  fork[(i+1)%N].down();
         else
                  fork[(i+1)%N].down();
                  fork[i].down();
         eat;
         fork[(i+1)%N].up();
         fork[i].up();
```

Solution 3

Allow at most N-1 philosophers at a time sitting at the table

```
semaphores fork[N] all initialized at 1
semaphore table initialized at N-1

Philosopher(i) :=
   Repeat
        think;
        table.down();
        fork[i].down();
        fork[(i+1)%N].down();
        eat;
        fork[(i+1)%N].up();
        fork[i].up();
        table.up()
```

Solution 4

Pick up 2 chopsticks only if both are free

- a philosopher moves to his/her eating state only if both neighbors are not in their eating states
  - → need to define a state for each philosopher
- if one of my neighbors is eating, and I'm hungry, ask them to signal me when they're done
  - → thus, states of each philosopher are: thinking, hungry, eating
  - → need condition variables to signal waiting hungry philosopher(s)

This solutoin very well fits with the features of monitors!

Solution 4


```
monitor DP
   status state[N] all initialized at thinking;
   condition self[N];

   Pickup(i) :=
        state[i] = hungry;
        test(i);
        if (state[i] != eating) then self[i].wait;

   Putdown(i) :=
        state[i] = thinking;
        test((i+1)%N);
        test((i-1)%N);

   test(i) :=
        if (state[(i+1)%N] != eating && state[(i-1)%N] != eating
            && state[i] == hungry)
        then    state[i] = eating;
                self[i].signal();
```

Software Transactional Memory

- Group together parts of the code that must look like atomic, in a way that is transparent, scalable and easy-to-use for the programmer
- Differently from monitors, the part of the code to group is not part of the definition of the objects, but is application dependent
- Differently form transactions in databases, the code can be any code, not just queries on the DB

Transaction: an atomic unit of computation (look like instantaneous and without overlap with any other transaction), that can access atomic objects.

→ *Assumption*: when executed alone, every transaction successfully terminates.

Program: set of sequential processes, each alternating transactional and non-transactional code (that both access base objects)

STM system: online algorithm that has to ensure the atomic execution of the transactional code of the program.

Software Transactional Memory

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

To guarantee efficiency, several transactions can be executed simultaneously (the so called *optimistic execution* approach), but then they must be totally ordered

→ not always possible (e.g., when there are different accesses to the same obj, with at least one of them that changes it)

→ commit/abort transactions at their completion point (or even before)

→ in case of abort, either try to re-execute or notify the invoking proc.

→ possibility of unbounded delay

Conceptually, a transaction is composed of 3 parts:

[READ of atomic reg's] [local comput.] [WRITE into shared memory]

The key issue is ensuring consistency of the shared memory

→ as soon as some inconsistency is discovered, the transaction is aborted

Implementation: every transition uses a local working space

- For every shared register: the first READ copies the value of the reg. in the local copy; successive READs will then read from the local copy
- Every WRITE modifies the local copy and puts the final value in the shared memory only at the end of the transaction (if it has not been aborted)

4 operations:
* `begin`$_T$`()` : initializes the local control variables
* `X.read`$_T$`()` , `X.write`$_T$`()` : as described above
* `try_to_commit`$_T$`()` : decides whether a non-aborted trans. can commit

A Logical Clock based STM system

Let T be a transaction; its *read prefix* is formed by all its successful READ before its possible abortion.

An execution is **opaque** if all committed transactions and all the read prefixes of all aborted transactions appear if executed one after the other, by following their real-time occurrence order.

We now present an atomic STM system, called *Transactional Locking 2* (TL2, 2006):

- CLOCK is an atomic READ/FETCH&ADD register initialized at 0
- Every MRMW register X is implemented by a pair of registers XX s.t.
  - XX.val contains the value of X
  - XX.date contains the date (in terms of CLOCK) of its last update
  - It is associated with a lock object (to guarantee MUTEX when updating the shared memory)
- For every transaction T, the invoking process maintains
  - lc(XX) : a local copy of the implementation of reg. X
  - read_set(T) : the set of names of all the registers read by T up to that moment
  - write_set(T) : the set of names of all the registers written by T up to that moment
  - birthdate(T) : the value of CLOCK(+1) at the starting of T

Idea: commit a transaction iff it could appear as executed at its birthdate time

Consistency:

- If T reads X, then it must be that XX.date < birthdate(T)
- To commit, all registers accessed by T cannot have been modified after T's birthdate (again, XX.date < birthdate(T))

A Logical Clock based STM system

SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

```
beginT() :=
    read_set(T), write_set(T) ← ∅
    birthdate(T) ← CLOCK+1

X.readT() :=
    if lc(XX)≠⊥ then return lc(XX).val
    lc(XX) ← XX
    if lc(XX).date ≥ birthdate(T) then ABORT
    read_set(T) ← read_set(T) ∪ {X}
    return lc(XX).val

X.writeT(v) :=
    if lc(XX)=⊥ then lc(XX) ← newloc
    lc(XX).val ← v
    write_set(T) ← write_set(T) ∪ {X}

try_to_commitT() :=
    lock all read_set(T) ∪ write_set(T)
    ∀ X ∈ read_set(T)
        if XX.date ≥ birthdate(T)
        then release all locks
             ABORT
    tmp ← CLOCK.fetch&add(1)+1
    ∀ X ∈ write_set(T)
        XX ← ⟨lc(XX).val , tmp⟩
    release all locks
    COMMIT
```

Remark: to avoid deadlock, there is a total order on the registers and locks are required by respecting this order (the deadlock is avoided as in Solution 1 of the Dining Philosophers)

Atomicity

When operations are made atomic (i.e., indivisible), programming concurrent applications becomes easier.

→ how can we turn a non-atomic execution into an atomic one (if possible)?

We have a set of n sequential processes p1,…,pn that access m concurrent objects X1,…,Xm by invoking operations of the form Xi.op(args)(ret).

When invoked by pj, the invocation Xi.op(args)(ret) is modeled by two events:

inv[Xi.op(args) by pj] and res[Xi.op(ret) to pj].

A **history** (or **trace**) is a pair $\widehat{H} = (H , <_H)$ where H is a set of events and $<_H$ is a total order on them

The *semantics* (of systems and/or objects) will be given as a set of traces.

A history is **sequential** if it is of the form inv res inv res … inv res inv inv inv … (where every res is the return operation of the immediately preceeding inv)

→ a sequential history can be represented as a sequence of operations

A history is **complete** if every inv is eventually followed by a corresponding res, **partial** otherwise.

Linearizability

Def.: a complete history $\hat{H}$ is **linearizable** if there exists a sequential history $\hat{S}$ s.t.

1. $\forall X . \hat{S}|_X \in$ semantics(X)
2. $\forall p . \hat{H}|_p = \hat{S}|_p$
3. If res[op] $<_H$ inv[op'], then res[op] $<_S$ inv[op']

Given an history $\hat{K}$, we can define a binary relation on events $\longrightarrow_K$ s.t. (op, op') $\in \longrightarrow_K$ if and only if res[op] $<_K$ inv[op']. We write op $\longrightarrow_K$ op' for denoting (op, op') $\in \longrightarrow_K$.

Hence, condition 3 of the previous Def. requires that $\longrightarrow_H \subseteq \longrightarrow_S$.

EXAMPLE: Let Q be a queue; let p1 and p2 be such that

```
           Q.enq(a)     Q.enq(b)             Q.deq -> b
p1  --|---------|---|---------|------------|-----------|------->
p2  --------|---------|------------|------------|-------------->
                 Q.enq(c)              Q.deq -> a
```

This corresponds to the history

inv[Q.enq(a) by p1] inv[Q.enq(c) by p2] res[Q.enq() to p1] inv[Q.enq(b) by p1] res[Q.enq() by p2] res[Q.enq() by p1] inv[Q.deq() by p2] inv[Q.deq() by p1] res[Q.deq(a) to p2] res[Q.deq(b) to p1]

It can be linearized as [Q.enq(a)() by p1] [Q.enq(b)() by p1] [Q.enq(c)() by p2] [Q.deq()(a) to p2] [Q.deq()(b) to p1]

Linearizability (cont.'d)

Now consider

```
               Q.enq(a)     Q.enq(b)                 Q.deq -> a
p1  --|---------|---|---------|------------|-----------|------->
p2  --------|---------|------------|------------|------------->
                     Q.enq(c)                Q.deq -> c
```

The corresponding history can still be linearized as

[Q.enq(c)() by p2] [Q.enq(a)() by p1] [Q.enq(b)() by p1] [Q.deq()(c) to p2] [Q.deq()(a) to p1]

By contrast, the following are not linearizable histories:

```
               Q.enq(a)     Q.enq(b)                 Q.deq -> a
p1  --|---------|---|---------|------------|-----------|------->
p2  --------|---------|------------|------------|------------->
                     Q.enq(c)                Q.deq -> a
```

```
               Q.enq(a)     Q.enq(b)                 Q.deq -> b
p1  --|---------|---|---------|------------|-----------|------->
p2  --------|---------|------------|------------|------------->
                     Q.enq(c)                Q.deq -> c
```

Compositionality of Linearizability

<u>Thm (compositionality):</u> $\widehat{H}$ is linearizable if $\widehat{H}|_X$ is linearizable, for all X involved in H

Proof (sketch):

For all X, let $\hat{S}_X$ be a linearization of $\widehat{H}|_X$

→ $\hat{S}_X$ defines a total order on the operations on X (call it $\longrightarrow_X$)

Let $\longrightarrow$ denote $\longrightarrow_H \cup \bigcup_{X \text{ in } H} \longrightarrow_X$ *(recall that a relation is a set of pairs, so here you take the union of all pairs of $\longrightarrow_H$ and of all $\longrightarrow_X$)*

It can be proved that $\longrightarrow$ is acyclic (it is a DAG).

Every DAG admits a topological order (i.e., a total order of its nodes that respects the edges)

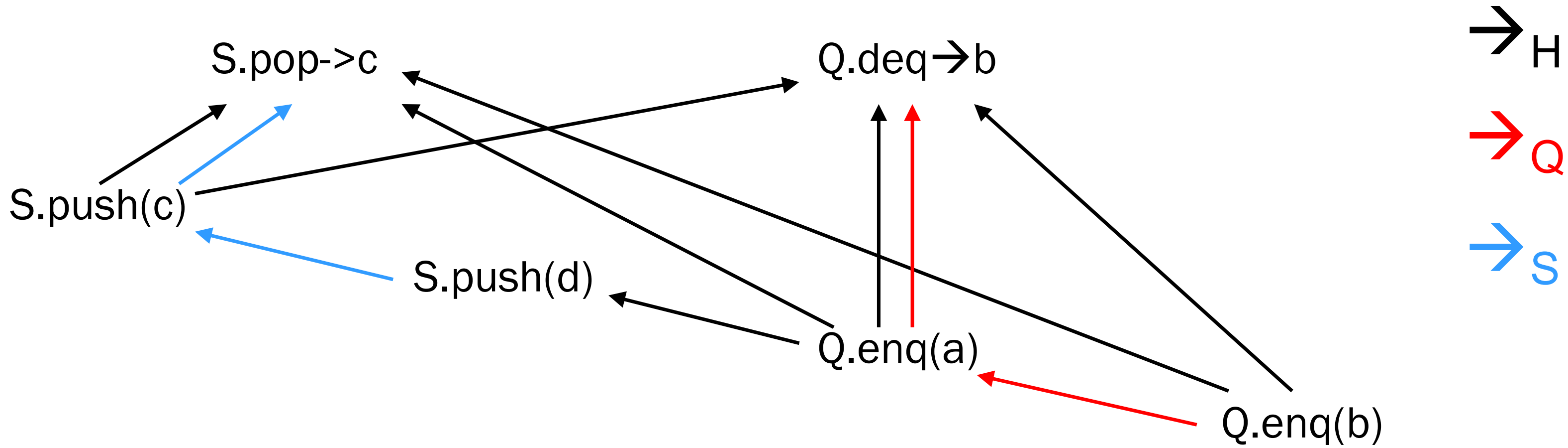
→ Let $\longrightarrow'$ denote a topological order for $\longrightarrow$
(with op1 $\longrightarrow'$ op2 $\longrightarrow'$ …)

We can then prove that the following is a linearization of $\widehat{H}$:

$\hat{S}$ = inv(op1) res(op1) inv(op2) res(op2) …

Example (of compositionality)

```
              Q.enq(a)        S.push(d)
p1  --|---------|-----|----------|--------------->
                     Q.enq(b)        Q.deq -> b
p2  --------|----------||---|------------|------->
                   S.push(c)         S.pop -> c
p3  --------|----------|------|------------|----->
```



TWO POSSIBLE LINEARIZATIONS:

1. Q.enq(b) , Q.enq(a) , S.push(d) , S.push(c) , Q.deq->b , S.pop->c
2. Q.enq(b) , Q.enq(a) , S.push(d) , S.push(c) , S.pop->c , Q.deq->b

An alternative to Atomicity

Sequential consistency

Let us define op $\longrightarrow_{proc}$ op' to hold whenever there exists a process p that issues both operations (with res[op] happening before inv[op']).

Def.: a complete history $\widehat{H}$ is **sequentially consistent** if there exists a sequential history $\hat{S}$ s.t.

1. $\forall$ X . $\hat{S}|_X \in$ semantics(X) *(like linearizability)*
2. $\forall$ p . $\widehat{H}|_p = \hat{S}|_p$ *(like linearizability)*
3. $\longrightarrow_{proc} \subseteq \longrightarrow_S$ *(in place of $\longrightarrow_H \subseteq \longrightarrow_S$)*

This is a more generous notion than linearizability.

EXAMPLE: Let $\widehat{H}$ be [Q.enq(a)() by p1] [Q.enq(b)() by p2] [Q.deq()(b) to p2]

→ not linearizable:
- the only possible linearization of $\widehat{H}$ is $\widehat{H}$ itself (because of cond.3)
- it violates the semantics of a queue (cond.1)

→ it is sequentially consistent, by swapping the first two actions, i.e. by considering $\hat{S}$ to be [Q.enq(b)() by p2] [Q.enq(a)() by p1] [Q.deq()(b) to p2]

An alternative to Atomicity

The problem with sequential consistency is that it is NOT compositional.

EXAMPLE

Consider the following two processes:

p1: Q.enq(a) ; Q'.enq(b') ; Q'.deq()→b'

p2: Q'.enq(a') ; Q.enq(b) ; Q.deq()→b

In isolation, both processes are sequentially consistent

However, no total order on the previous 6 operations respects the semantics of a queue:

- If p1 receives b' from Q'.deq, we have that Q'.enq(a') must arrive after Q'.enq(b')
- To respect $\longrightarrow_{proc}$, also the remaining behaviour of p2 must arrive after
- Hence, Q.enq(a) arrived before Q.enq(b) and so it is not possible for p2 to receive b from its Q.deq

Hence, we have two histories that are sequentially consistent but whose composition cannot be sequentially consistent → no compositionality!