Shared-memory systems.

Linearizability. Implementations.

Note: for the next few weeks, all the systems we talk about are **asynchronous**, with **halting failures**.
References for today: [HW90], [AW98], Chapter 9 or [Lyn96], Chapters 9 and 13.1.

In a shared-memory system, processes communicate with one another by accessing shared data structures (of various types) called objects.

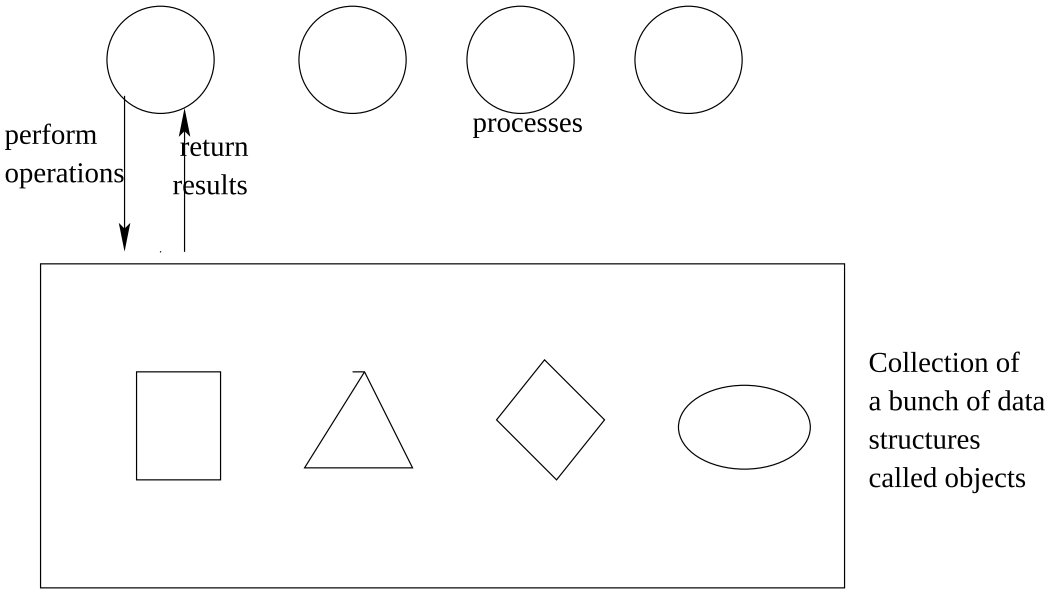


Figure 1: shared-memory system

Shared Objects: State of the object represents data stored in it
Equipped with a set of operations that can be performed
Each operation changes the state of object and returns a response to the process

Example 1, (Read/write) register: state $\in \mathbf{N}$, two operations:
read() : returns the state without modify it
write(v) : changes state to v and returns ack.

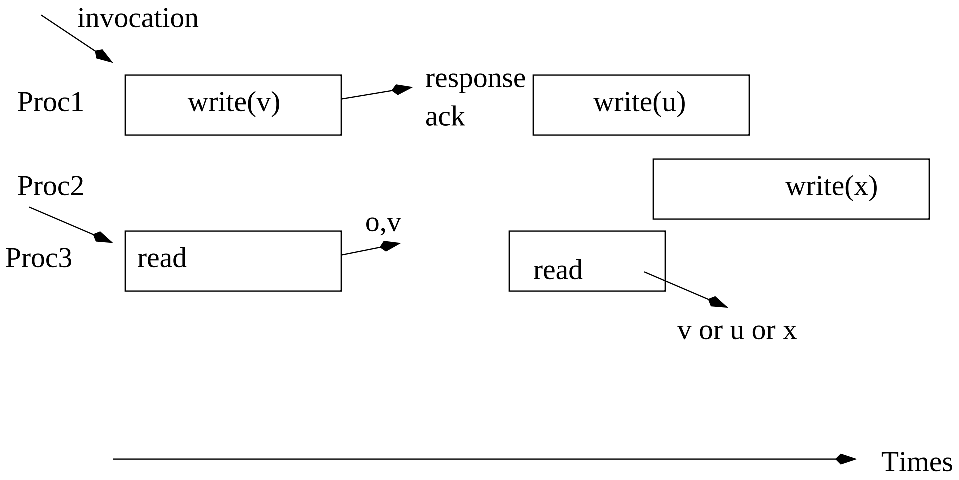
Example2, FIFO Queue: state is a finite sequence of natural numbers
enq(v) : appends v to the end of of sequence and returns ack deq() : removes and returns first element of the sequence (or returns null if sequence is empty).

Sequential Specification: Describes how object behaves if it is accessed by one operation at a time.
Need additional info to describe how objects behave when is is accessed by concurrent operations.

eg. One might say objects can return arbitrary responses if it is ever accessed concurrently.

1. very weak condition
2. easy to implement
3. hard to use (need to have mutual excusion in order to make use of the object in such a way that no two ops are concurrent).

But we want to allow concurrent operations. Consider a register being accessed by 3 processes as shown below. The outcomes of the operations should not be uniquely specified, but they should be constrained. For example, it would be strange if the first process returned any value other than 0 (the initial value of the register) or v. The second read should return either v, u or x, depending on whether we think of the write(u) and write(x) operations as having taken place before the read or after. (Notice that the second read shouldn't really return 0, since the write(v) operation should definitely happen before the second read.) We formalize these "reasonable" constraints using the idea of linearizability.



Linearizability: This is a very good way to describe how an object is supposed to behave when accessed concurrently, because linearizable objects are relatively easy for programmers to use.

An object is linearizable if for any history there exists a total ordering of all operations (except maybe some incomplete ones) s.t. if these operations were done sequentially in that order, the responses would be same.

Furthermore : if operation 1 completes before operation2 begins then op1 should come before op2 in the total order.

This condition together wuth sequential specification completely describes behaviour of shared objects. Linearizable objects are easy to use.

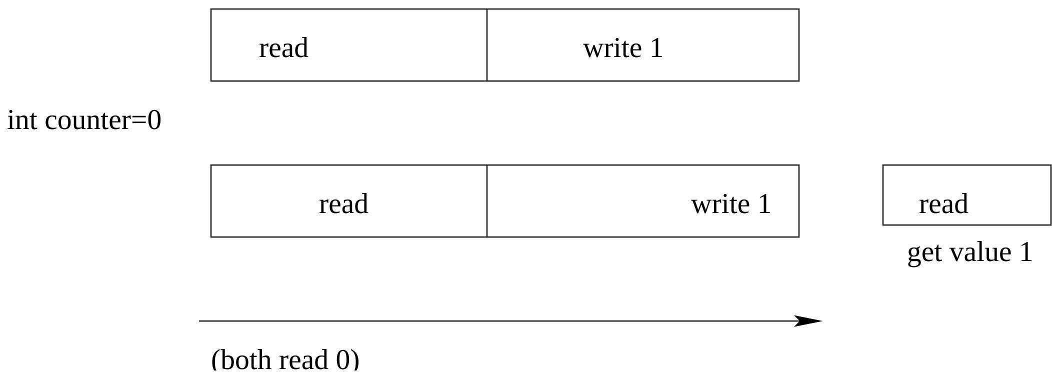
There are lots of other consistency conditions lying somewhere between the strong condition of linearizability and the weak condition mentioned above (no concurrency allowed). Eg. serializability, hybrid consistency, etc. But we'll focus on linearizable objects.

Question: Given read/write registers, how can we implement a (linearizable) counter object.

The state of a counter stores a natural number, and it is equipped with two operations :
INC : increments state by 1, returns ack
READ : returns state without changing it

Wrong Solution: Use a single shared register and implement the INC command as follows:
$v \leftarrow$ read register
write $v + 1$ into register
To READ the counter, just read the register.

Consider an execution of two concurrent INCs where both INCs reads precede both INC's writes. Then both INCs will write the value 1 into the register. When both INCs are complete, somebody READs the counter.



Both INCs must be linearized before the counter READ operation—so that READ should return value 2 (and it doesn't!).

Correct Solution: Implementation from n registers $A[1....n]$ for n processes

Code for process i:
INC :
  $v \leftarrow A[i]$
  $A[i] \leftarrow v + 1$
$READ$:
  for $j \leftarrow 1$ to n
    read $A[j]$ and sum them up
  return sum;

We want to show this is a linearizable implementation.

It suffices to specify, for any execution, a linearization point for each operation. This means that we will think of the operation as taking place at that exact moment. We shall choose the linearization point of each operation to be within the interval of time from the operation's invocation to the operation's response. Then we show that if each operation took place instantaneously at its linearization point, the responses of all operations would be the same as they are in the execution we're considering.

Consider any execution.

The linearization points for each INC is the moment where the process performs the write to its register.

Let r_i be the value of $A[i]$ at the start of the read operation.
Let r_i' be the value of $A[i]$ at the end of the read operation.
Let r_i'' be the value of $A[i]$ the READ operation sees when it reads A[i].
Then we have $r_i <= r_i' <= r_i''$, since the value stored in $A[i]$ never decreases.

So, $\sum_{i=1}^{n} r_i \leq \sum_{i=1}^{n} r_i'' \leq \sum_{i=1}^{n} r_i'$. When the READ begins, the sum of $A[1..n]$ is $\sum_{i=1}^{n} r_i$ and when the READ ends, the sum of $A[1..n]$ is $\sum_{i=1}^{n} r_i'$. Because the sum of $A[1..n]$ only increases by 1 at a time, the sum of $A[1..n]$ must be equal to $\sum_{i=1}^{n} r_i''$, which is the value returned by the READ, at some time during the READ. Choose that moment as the point at which the READ is linearized.
This choice of linearization points ensures that each READ returns an output that is equal to the number of INCs that have earlier linearization points (assuming the initial value of the counter was 0). Now imagine performing the operations of the execution one by one in the order specified by the linearization points. Every operation would return exactly the same result in this imaginary execution as it does in the real execution we started with.

Solving Consensus using Shared memory: Strong Shared Objects can be used to solve consensus. For example, the Compare-and-Swap object, which is defined as follows.
The state stores a natural number
There is one operation, $C\&S$ (old,new), which has the following effect.

```
C&S(old,new)
    temp ←
    if state = old then
        state ← new
    return temp
```

Here's the code for process i to execute in the consensus algorithm. The algorithm uses one shared Compare-and-Swap object, initially in state $\perp$.

```
Consensus(input value)
    C&S (⊥, input value for consensus)
    if return value is ⊥
        output your input
    else
        output value returned by C&S operation
```

Why does this solve consensus?

Every proc will return the input value of first process to access the $CandS$ object. Once the value of $CandS$ object changes, it will never go back to $\perp$, so only the first $CandS$ operation will successfully change the value of object. Everyone else will return the value that was written into the object by that process.