TDC1:
Safety and liveness

Methodology

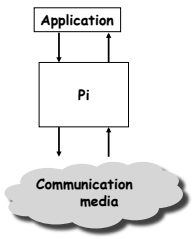
- Define models (tractability, realism)
- Devise abstractions for the system design (convenience, efficiency)
- Devise algorithms and determine complexity bounds

Basic abstractions

- *Process* abstraction – an entity performing independent computation
- Communication
  - ✓Message-passing: *channel* abstraction
  - ✓Shared memory: *objects*

Processes

- Automaton Pi (i=1,...,N):
  - ✓States
  - ✓Inputs
  - ✓Outputs
  - ✓Sequential specification

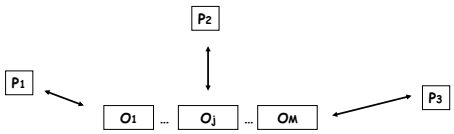


Algorithm = {P_1,…,P_N}

- Deterministic algorithms
- Randomized algorithms

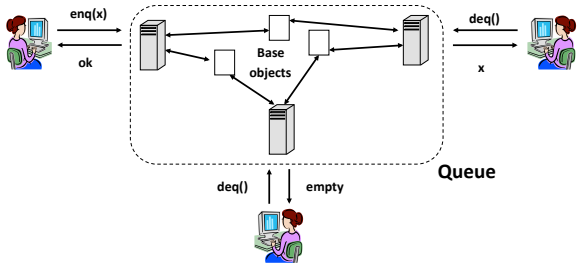
Shared memory

- Processes communicate by applying operations on and receiving responses from *shared objects*
- A shared object instantiates a state machine
  - ✓States
  - ✓Operations/Responses
  - ✓Sequential specification
- Examples: read-write registers, T&S,C&S,LLSC,…



Implementing an object

Using *base* objects, create an illusion that an object O is available



Correctness

What does it mean for an implementation to be correct?

- Safety ≈ nothing bad ever happens
  - ✓Can be violated in a finite execution, e.g., by producing a wrong output or sending an incorrect message
  - ✓What the implementation is allowed to output
- Liveness ≈ something good eventually happens
  - ✓Can only be violated in an *infinite* execution, e.g., by never producing an expected output
  - ✓Under which condition the implementation does return

Traces

A system trace is a sequence of events

- ✓E.g., actions that processes may take

Σ – event alphabet

- ✓ E.g., all possible actions
- ✓A finite sequence of actions determines a system state

A property P is a subset of Σ*

An implementation satisfies P if every its trace is in P

Safety properties

P is a safety property if:

- P is prefix-closed: if σ is in P, then each prefix of σ is in P
- P is limit-closed: for each sequence of traces σ_0, σ_1, σ_2,…, such that each σ_i is a prefix of σ_{i+1} and each σ_i is in P, the limit trace σ is in P

(Enough to prove safety for all finite traces)

Liveness properties

P is a liveness property if for every finite σ in Σ* has an extension in P

(Enough to prove liveness for all infinite traces)

Homework I

- Safety? Liveness?

  Processes propose values and decide on values:

  $\Sigma = U_{i,v}\{propose_i(v), decide_i(v)\} \cup \{protocol\ actions\}$

  - ✓Every decided value was previously proposed
  - ✓No two processes decide differently
  - ✓Every correct (taking infinitely many steps) process eventually decides
  - ✓No two correct processes decide differently
- Show that every property is a conjunction of safety and liveness

Implementing a concurrent queue

What *is* a concurrent FIFO queue?

✓FIFO means strict temporal order
✓Concurrent means ambiguous temporal order

When we use a lock…

```
deq()

  lock.lock();
    if (tail = head)
       x := empty;
    else
       x := items[head];
       head++;
  lock.unlock();
  return x;
```

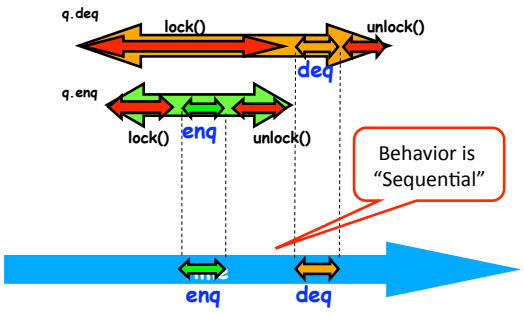

Intuitively…

```
deq()

  lock.lock();
    if (tail = head)
       x := empty;
    else
       x := items[head];
       head++;
  lock.unlock();
  return x;
```

All modifications of queue are done in mutual exclusion

We describe
the concurrent via the sequential



Linearizability: A Safety Property

- Each complete operation should
  - ✓"take effect"
  - ✓Instantaneously
  - ✓Between invocation and response events
- Object is correct if every its "sequential behavior" is correct

Why not using one lock?

- Simple – automatic transformation of the sequential code
- Correct – linearizability for free
- In the best case, starvation-free: if the lock is "fair" and every process cooperates, every process makes progress
- Not robust to failures/asynchrony
  - ✓ Cache misses, page faults, swap outs
- Fine-grained locking?
  - ✓ Complicated/prone to deadlocks
- This course is about lock-free computing

Liveness properties

- *Deadlock-free:*
  - ✓If every process cooperates (takes enough steps), some process makes progress
- *Starvation-free:*
  - ✓If every process cooperates, every process makes progress
- *Non-blocking:*
  - ✓Some active process makes progress
- *Wait-free:*
  - ✓Every active process makes progress
- *Obstruction-free:*
  - ✓Every process makes progress if it executes in isolation

Homework II

- Implement enq() in the single-lock queue algorithm
- Make it bounded, assuming that the queue returns "full" when the queue size reaches the bound
- What are the relations (weaker stronger) between various progress properties?