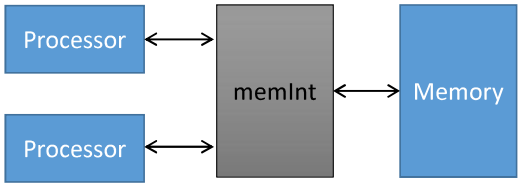
Ch5: Caching Memory



Separate definitions for Interface and Memory

~~**Modules**~~
~~AsyncInterface~~
~~FIFO~~
~~BoundedBuffer~~

- Abstract away memInt details (e.g. handshake)
  - Send and Receive single step operations
    - Operations change memInt in some unspecified way

    CONSTANT Send(_, _, _, _)

    ASSUME ∀ p,d, miOld, miNew: Send(p,d, miOld, miNew) ∈ BOOLEAN

- Constant Parameters: Proc, Adr, Val
- Values sent over interface

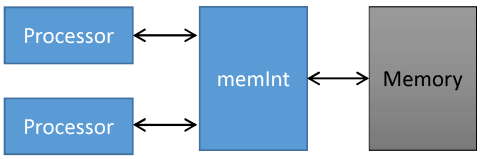
  Mreq == [op:{"Rd"}, adr:Adr] U [op:{"Wr"}, adr:Adr, val:Val]

  NoVal == CHOOSE v : v ∉ Val

Memory Interface

```
---------- MODULE MemoryInterface ---------
  VARIABLE memInt
  CONSTANTS  Send(_, _, _, _),
             Reply(_, _, _, _),
             InitMemInt,
             Proc,
             Adr,
             Val

  ASSUME \A p, d, miOld, miNew :
        /\ Send(p,d,miOld,miNew)  \in BOOLEAN
        /\ Reply(p,d,miOld,miNew) \in BOOLEAN

  ----------------------------------------------------------
  MReq == [ op : {"Rd"},   adr: Adr              ]
       \cup [ op : {"Wr"},  adr: Adr, val : Val  ]

  NoVal == CHOOSE v : v \notin Val
  ==================================
```

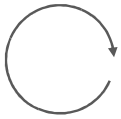
Linearizable Memory



- Each processor may make one request at a time
- Memory access occurs any time between a request and the response.

Variables:

- mem[adr] = val
- *buf[p]* holds request or response record
  [op |-> "WR", adr |-> a, val |-> v]
- ctl[p]



"rdy"	memory sets when it can accept request	initialize
"busy"	processor sets when buffer has request	*Req(p)*
"done"	memory has changed mem[a] = v	*Do(p)*
"rdy"	*memory sets when sends response*	Rsp(p)

---------- MODULE InternalMemory ----------
EXTENDS MemoryInterface
VARIABLES mem, ctl, buf

```
IInit == /\ mem  ∈  [Adr->Val]
         /\ ctl = [p  ∈  Proc |-> "rdy"]
         /\ buf = [p  ∈  Proc |-> NoVal]
         /\ memInt  ∈  InitMemInt

TypeInvariant ==
 /\ mem  ∈  [Adr->Val]
 /\ ctl  ∈  [Proc -> {"rdy", "busy","done"}]
 /\ buf  ∈  [Proc -> MReq  U  Val  U  {NoVal}]
```

```
Req(p) == /\ ctl[p] = "rdy"
          /\  ∃ req ∈  MReq :
              /\ Send(p, req, memInt, memInt')
              /\ buf' = [buf EXCEPT ![p] = req]
              /\ ctl' = [ctl EXCEPT ![p] = "busy"]
          /\ UNCHANGED mem
```


```
Do(p) ==
       /\ ctl[p] = "busy"
       /\ mem' = IF buf[p].op = "Wr"
          THEN [mem EXCEPT ![buf[p].adr] = buf[p].val]
          ELSE mem
        /\ buf' = [buf EXCEPT ![p] = IF buf[p].op = "Wr"
                        THEN NoVal
                        ELSE mem[buf[p].adr]]
       /\ ctl' = [ctl EXCEPT ![p] = "done"]
       /\ UNCHANGED memInt
```

```
Rsp(p) == /\ ctl[p] = "done"
          /\ Reply(p, buf[p], memInt, memInt')
          /\ ctl' = [ctl EXCEPT ![p]= "rdy"]
          /\ UNCHANGED <<mem, buf>>
```

```
INext ==  ∃ p ∈  Proc: Req(p) \/ Do(p) \/ Rsp(p)

ISpec == IInit  /\  [][INext]<memInt, mem, ctl, buf>
```

THEOREM ISpec => []TypeInvariant

==========

---------- MODULE Memory ----------
EXTENDS MemoryInterface
Inner(mem, ctl, buf) == INSTANCE InternalMemory
Spec == ∃ mem, ctl, buf : Inner(mem, ctl, buf)!ISpec
==========

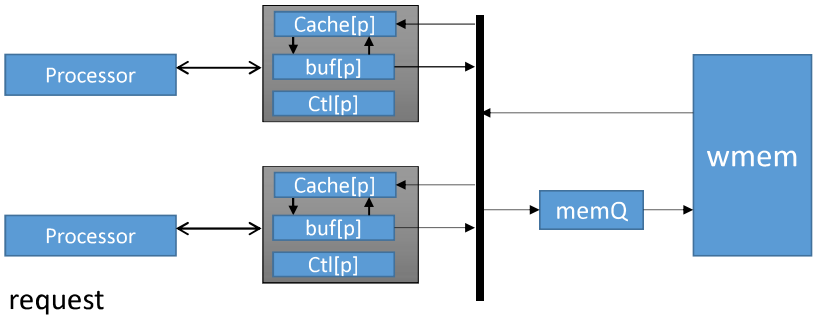
Recursive Function Definitions

- *fact[n] = IF n=0 THEN 1 ELSE n*fact[n-1]*
- In TLA+

  ~~*fact == [n\in Nat |->IF n=0 THEN 1 ELSE n* fact[n-1]]*~~

  fact [n\in Nat] == IF n=0 THEN 1 ELSE n fact[n-1]*

Write-Through Cache



- Write request
  - Processor action
    - Write to cache[p]
    - Put request into memQ
  - memQ action
    - store head of queue into wmem
- Read request
  - Processor action
    - If cache hit, return value in cache
    - if cache miss
      - Add request to memQ
      - Set ctl[p] to new *waiting* state
  - memQ action
    - Read last write to the address in memQ or wmem

Write-Through Cache

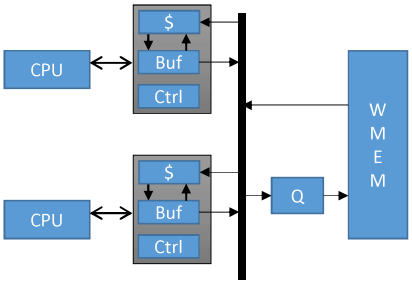
- Eviction: *Evict(p)*
  - Normally, when cache full, but doesn't affect algorithm correctness, just performance
  - Allow anytime, except when just put in cache

- Representing the cache and queue
  - memQ: sequence of pairs: *<p, req>*
  - *Cache[p][a]:* copy in cache *p* for address *a* or *NoVal*

- Operations

Req(p)	MemQWr
Rsp(p)	MemQRd
RdMiss(p)	vmem
DoRd(p)	
DoWr(p)	
Evict(p,a)	

```
------------------ MODULE WriteThroughCache -----------
EXTENDS Naturals, Sequences, MemoryInterface
VARIABLES wmem, ctl, buf, cache, memQ
CONSTANT QLen
ASSUME (QLen ∈ Nat) /\ (QLen > 0)
M == INSTANCE InternalMemory WITH mem <-wmem
-----------------------------------------------------------------
Init == /\ M!Init
        /\ cache = [p ∈ Proc |-> [a ∈ Adr |-> NoVal] ]
        /\ memQ = << >>

TypeInvariant ==
  /\ wmem    ∈  [Adr -> Val]
  /\ ctl     ∈ [Proc -> {"rdy", "busy", "waiting", "done"}]
  /\ buf     ∈ [Proc -> MReq U Val  U {NoVal}]
  /\ cache   ∈ [Proc -> [Adr -> Val U {NoVal}]]
  /\ memQ    ∈  Seq(Proc X MReq)

Coherence == ∀ p, q  ∈  Proc, a  ∈  Adr :
               (NoVal ∉ {cache[p][a], cache[q][a]})
                   => (cache[p][a]=cache[q][a])
-----------------------------------------------------------------
Req(p) == M!Req(p) /\ UNCHANGED <<cache, memQ>>
Rsp(p) == M!Rsp(p) /\ UNCHANGED <<cache, memQ>>
```



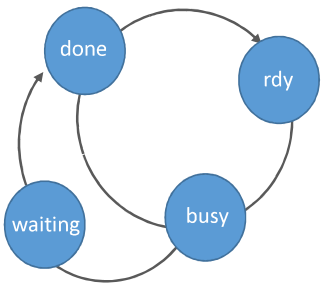
If two caches include a valid entry for the same address, the value must be the same

Bottom of spec

```
Evict(p,a) == /\ (ctl[p] = "waiting") => (buf[p].adr # a)
              /\ cache' = [cache EXCEPT ![p][a] = NoVal]
              /\ UNCHANGED <<memInt, wmem, buf, ctl, memQ>>

Next == \/ ∃ p ∈ Proc :
             \/ Req(p) \/ Rsp(p)
             \/ RdMiss(p) \/ DoRd(p) \/ DoWr(p)
             \/ ∃ a ∈ Adr : Evict(p, a)
        \/ MemQWr
        \/ MemQRd

Spec ==
 Init /\ [][Next]_<<memInt, wmem, buf, ctl, cache, memQ>>
----------------------------------------------------------------
THEOREM Spec => [](TypeInvariant /\ Coherence)
----------------------------------------------------------------
LM == INSTANCE Memory
THEOREM Spec => LM!Spec
================================================================
```

h

```
RdMiss(p) ==  /\ (ctl[p] = "busy") /\ (buf[p].op = "Rd")
              /\ cache[p][buf[p].adr] = NoVal
              /\ Len(memQ) < QLen
              /\ memQ' = Append(memQ, <<p, buf[p]>>)
              /\ ctl' = [ctl EXCEPT ![p] = "waiting"]
              /\ UNCHANGED <<memInt, wmem, buf, cache>>
DoRd(p) ==
 /\ ctl[p] ∈ {"busy","waiting"}
 /\ buf[p].op = "Rd"
 /\ cache[p][buf[p].adr] # NoVal
 /\ buf' = [buf EXCEPT ![p] = cache[p][buf[p].adr]]
 /\ ctl' = [ctl EXCEPT ![p] = "done"]
 /\ UNCHANGED <<memInt, wmem, cache, memQ>>

DoWr(p) ==
 LET r == buf[p]
 IN  /\ (ctl[p] = "busy") /\ (r.op = "Wr")
     /\ Len(memQ) < QLen
     /\ cache' = [q  ∈  Proc |->
                IF (p=q)  \/  (cache[q][r.adr]#NoVal)
                THEN [cache[q] EXCEPT ![r.adr] = r.val]
                ELSE cache[q] ]
    /\ memQ' = Append(memQ, <<p, r>>)
    /\ buf' = [buf EXCEPT ![p] = NoVal]
    /\ ctl' = [ctl EXCEPT ![p] = "done"]
    /\ UNCHANGED <<memInt, wmem>>
```



```
vmem  ==
 LET f[i  ∈  0 .. Len(memQ)] ==
     IF i=0 THEN wmem
          ELSE IF memQ[i][2].op = "Rd"
               THEN f[i-1]
               ELSE [f[i-1] EXCEPT ![memQ[i][2].adr] =
                                   memQ[i][2].val]
 IN f[Len(memQ)]

MemQWr == LET r == Head(memQ)[2]
       IN  /\ (memQ # << >>)  /\   (r.op = "Wr")
           /\ wmem' = [wmem EXCEPT ![r.adr] = r.val]
           /\ memQ' = Tail(memQ)
           /\ UNCHANGED <<memInt, buf, ctl, cache>>

MemQRd ==
 LET p == Head(memQ)[1]
      r == Head(memQ)[2]
 IN  /\ (memQ # << >> ) /\ (r.op = "Rd")
     /\ memQ' = Tail(memQ)
     /\ cache' = [cache EXCEPT ![p][r.adr] = vmem[r.adr]]
     /\ UNCHANGED <<memInt, wmem, buf, ctl>>
```

memQ ∈ Seq(Proc X MReq)

Skipping

- 5.7 Invariance
- 5.8 Proving Implementation
- Ch 6