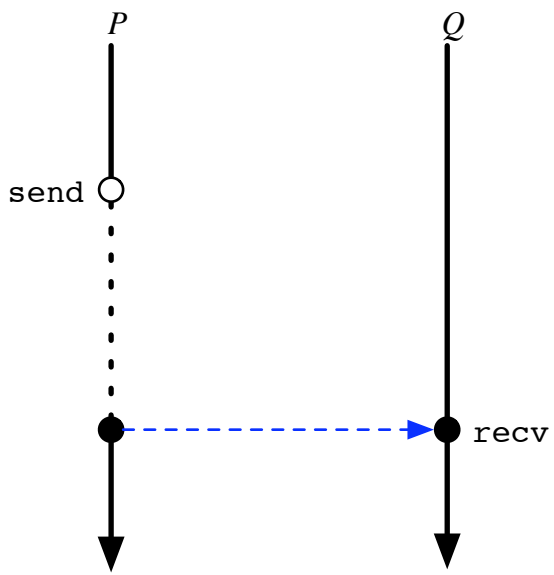
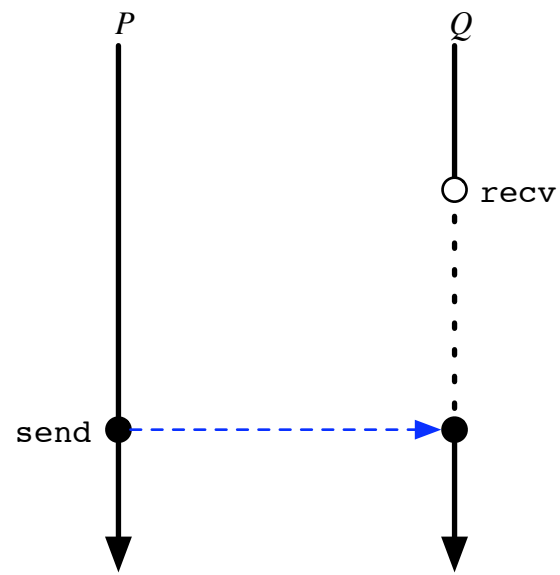
Thread primitives

```
type thread_id

val spawn : (unit -> unit) -> thread_id
val spawnc : ('a -> unit) -> 'a -> thread_id

val exit : unit -> 'a

val tidToString : thread_id -> string
```

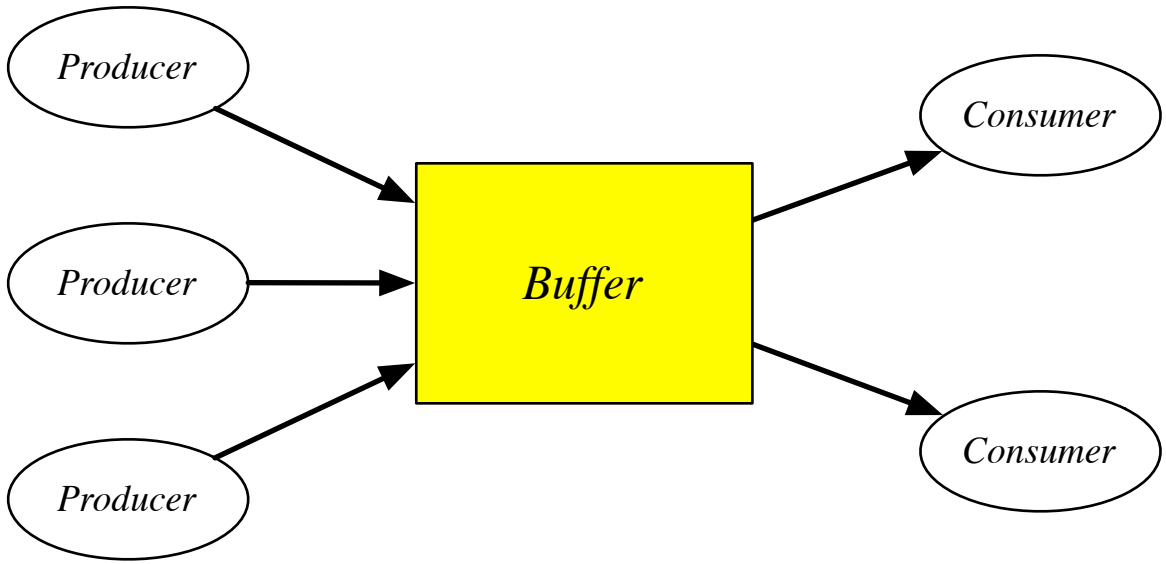
Synchronous channels

```
type 'a chan

val channel : unit -> 'a chan

val send : ('a chan * 'a) -> unit
val recv : 'a chan -> 'a
```





Producer/Consumer Buffer



Producer/Consumer buffer using Asynchronous Message Passing

```
signature BUFFER =
  sig
    type 'a buffer

    val buffer : unit -> 'a buffer

    val insert : ('a buffer * 'a) -> unit
    val remove : 'a buffer -> 'a
  end

structure Buffer :> BUFFER =
  struct
    ...
  end
```

Producer/Consumer buffer using Synchronous Message Passing

```
datatype 'a buffer = BUF of {
    insCh    : 'a chan,
    remCh    : 'a chan
  }

fun insert (BUF{insCh, ...}, v) = send(insCh, v)

fun remove (BUF{remCh, ...}) = recv remCh
```

Producer/Consumer buffer using Synchronous Message Passing

```
fun buffer () = let
      val insCh = chan()
      val remCh = chan()
      fun empty () = full (recv insCh)
      and full x = empty (recv remCh)
      in
        spawn empty;
        BUF{
            insCh = insCh,
            remCh = remCh
          }
      end
```

Selective communication

```
type 'a event

val sendEvt : ('a chan * 'a) -> unit event
val recvEvt : 'a chan -> 'a event

val wrap    : ('a event * ('a  -> 'b)) -> 'b event
val select  : 'a event list -> 'a
```

Unbounded buffer using Synchronous Message Passing

```
datatype 'a buffer = BUF of {
    insCh     : 'a chan,
    remCh     : 'a chan
  }

fun insert (BUF{insCh, ...}, v) = send(insCh, v)

fun remove (BUF{remCh, ...}) = recv remCh
```

Unbounded buffer using Synchronous Message Passing

```
fun buffer () = let
      val insCh = chan()
      val remCh = chan()
      fun loop q = (case next q
            of NONE =>
                  loop (ins (q, recv insCh))
             | SOME(x, q') => select [
                  wrap(recvEvt insCh,
                    fn y => loop (ins (q, y))),
                  wrap(sendEvt(remCh, x),
                    fn () => loop q')
                ]
           (* end case *))
      in
        spawnc loop empty;
        BUF{
            insCh = insCh,
            remCh = remCh
          }
      end
```

Implementation of FIFO Buffers

```
datatype 'a =
  BUF of {front : 'a list, rear : 'a list}

val empty = BUF{front=[], rear=[]}

fun ins (BUF{front, rear}, x) =
      BUF{front=front, rear=x::rear}

fun next (BUF{front=[], rear=[]}) = NONE
  | next (BUF{front=[], rear}) =
      next (BUF{front=List.rev rear, rear=[]})
  | next (BUF{front=x::r, rear}) =
      SOME(x, BUF{front=r, rear=rear})
```

Other synchronous operations

```
val joinEvt : thread_id -> unit event

val timeOutEvt : Time.time -> unit event

val atTimeEvt : Time.time -> unit event
```

Other CML communication mechanisms

- Mailboxes --- asynchronous channels
- Multicast channels
- iVars and mVars --- synchronous memory