Message Passing

- No shared memory
  - Can’t use simple semaphores, condition variables
  - Can’t use shared buffers, producer/consumer
- Communication based on *message passing*
  - Process A on machine 1 sends message to process B on machine 2 (over the network)
  - How does it get there? [we will ignore this]

Physical Reality of Networks

- Networks are unreliable
  - Messages are divided into packets
  - Packets can get lost
  - Packets can arrive out of order
  - Receiver can get overloaded

Define a new abstraction

- Analogous to abstractions in operating systems
  - Process -- abstraction of a processor
  - Virtual memory -- abstraction of unlimited memory
  - Files -- abstraction of disk
- Want to abstract communication network
  - Don't want to worry about lost messages, wrong ordering, overflow, etc.
  - **Channel** -- abstraction of point to point, reliable communication link

Send and Receive

- Send(channel, exprs);
  - exprs can be any expression (i.e., an r-val)
- Receive(channel, args);
  - args must be an l-val
- Notes:
  - Channel handles reliability
    - must be implemented by network protocols
  - Message may have to be buffered
    - depends on semantics of send/receive
  - Requires synchronization
  - Send, Receive can be OS kernel primitives or can be library primitives (e.g., MPI)

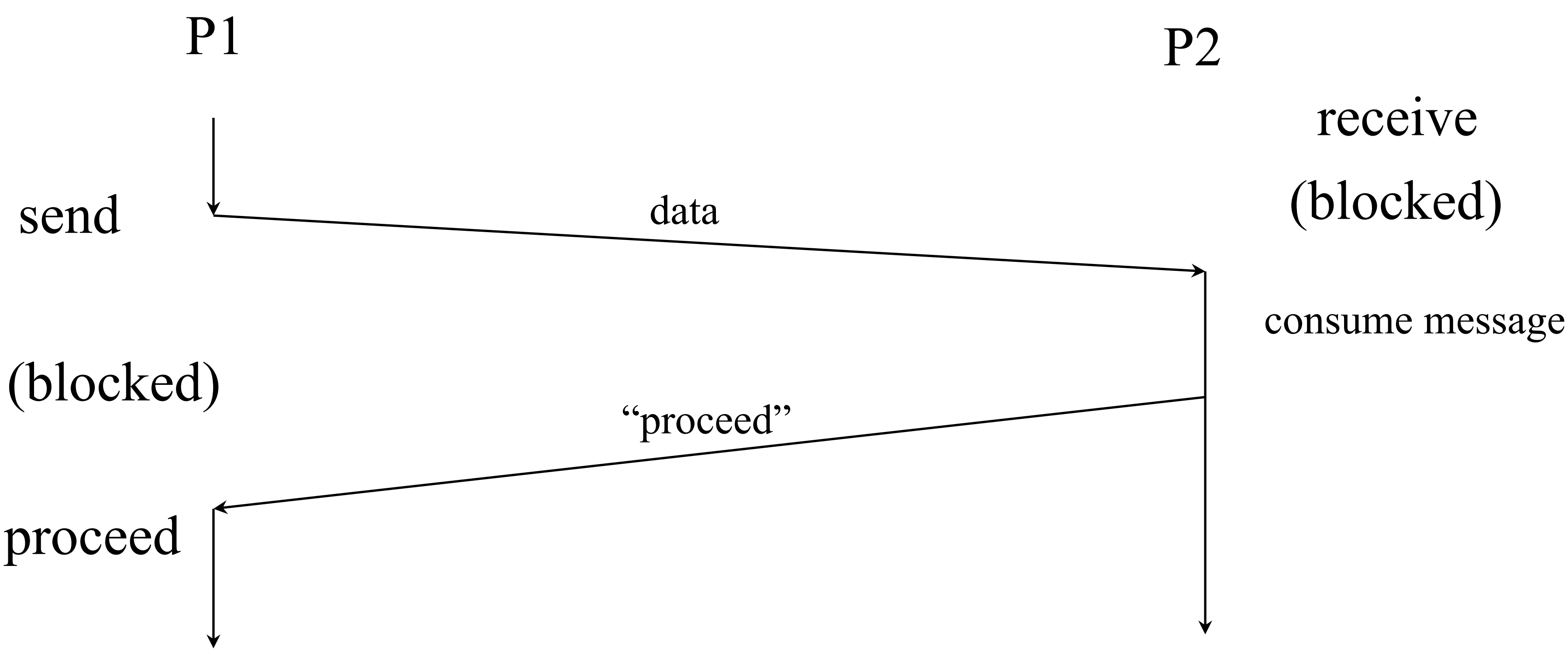
Send and Receive

- Notes, continued:
  - Special case: both *exprs* and *args* are the empty set and on single machine; in this case:
    - Send is equivalent to V(s)
    - Receive is equivalent to P(s)
    - Number of pending messages is equivalent to the value of *s*

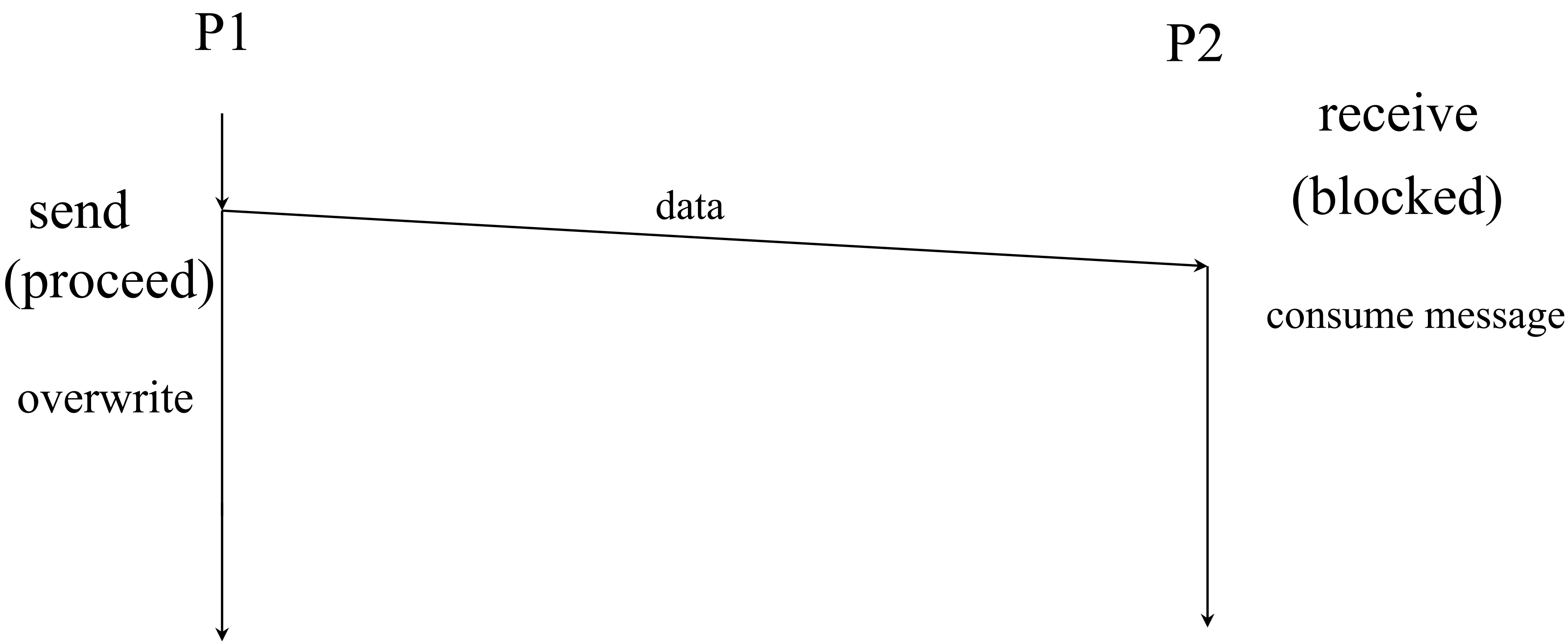
Semantics of Send and Receive

- Can be blocking or nonblocking
  - Also called “synchronous” and “asynchronous”
  - Remember:
    - procedure call is blocking
    - thread creation is non-blocking
  - Send, Receive both have blocking and non-blocking implementations

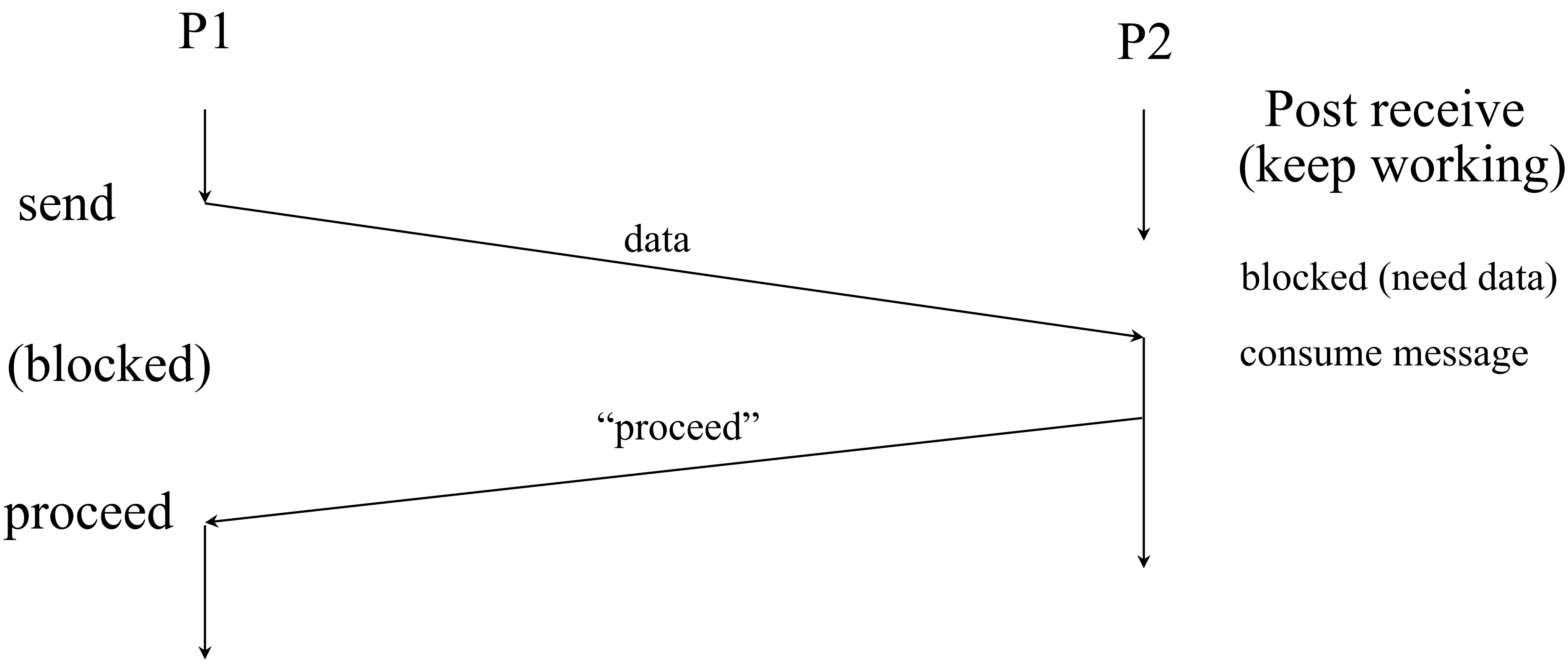
Picture of Blocking Send, Blocking Receive



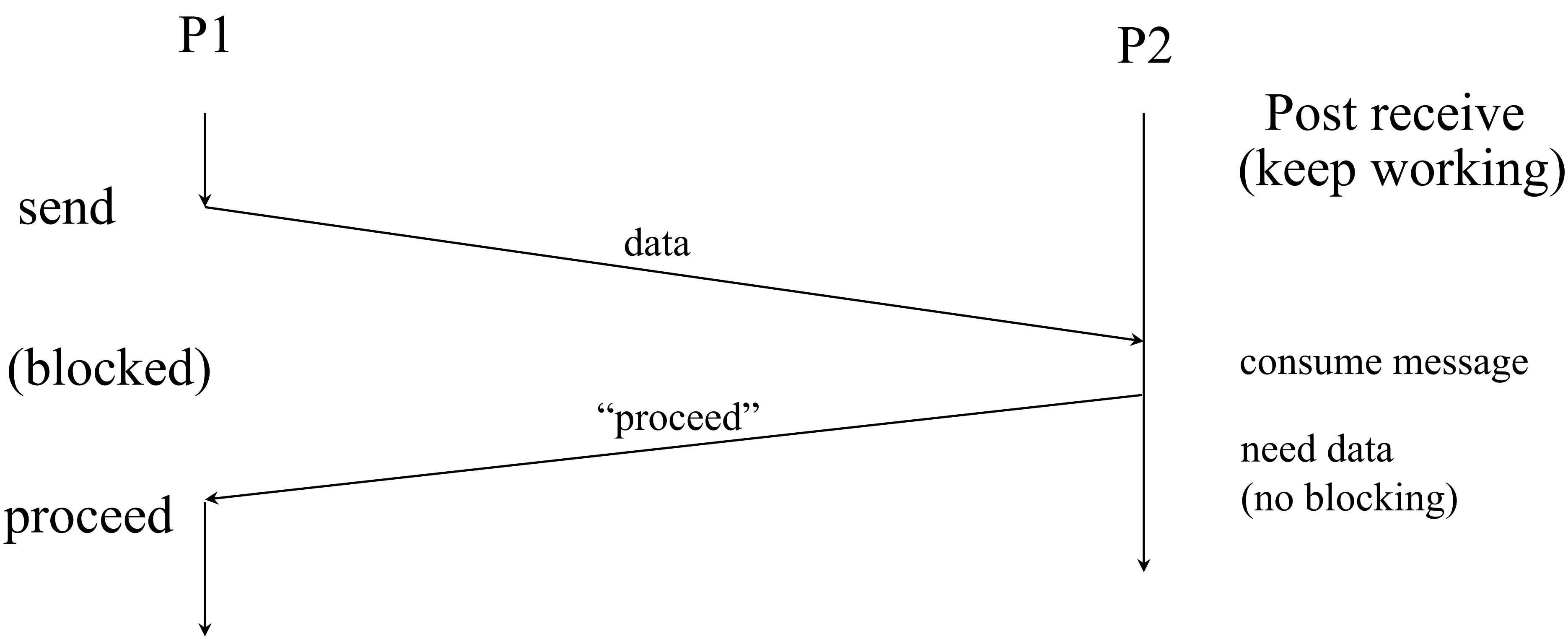
Picture of Non-Blocking Send, Blocking Receive



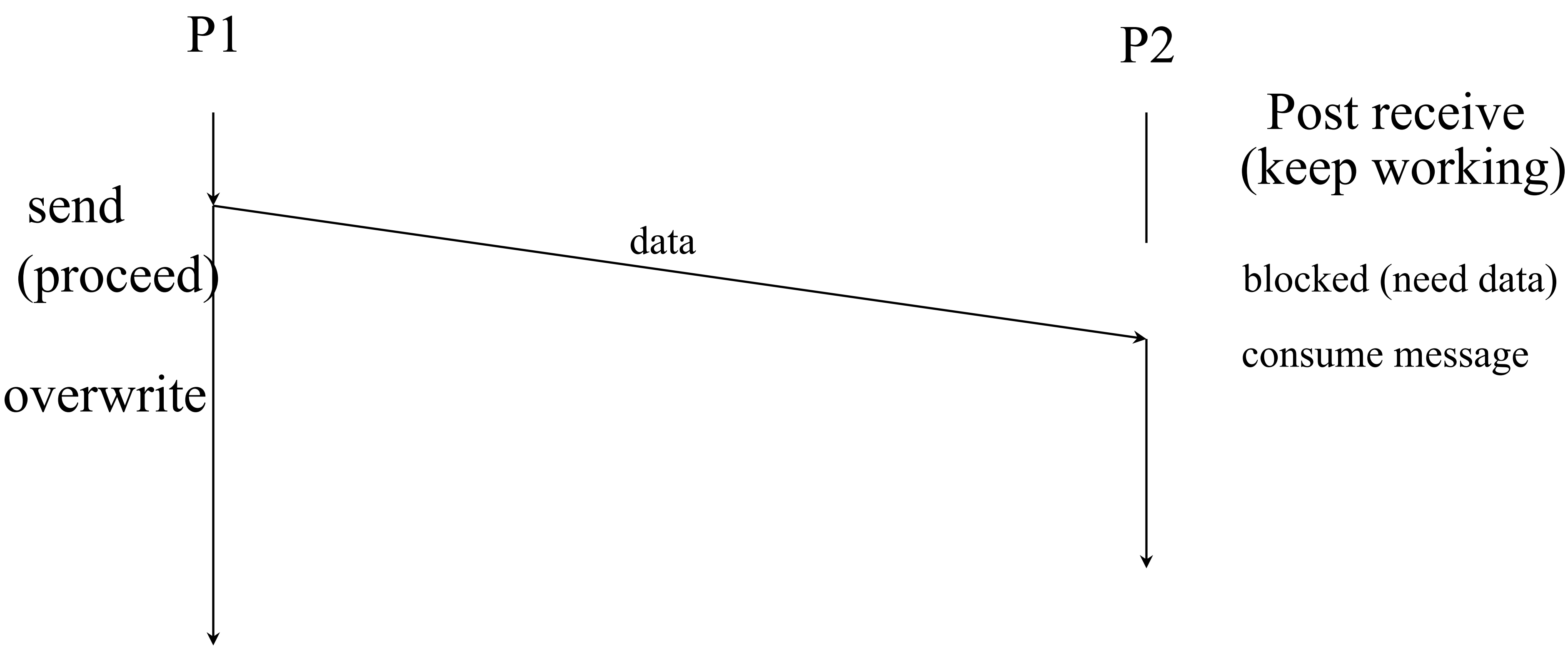
Picture of Blocking Send, Non-Blocking Receive



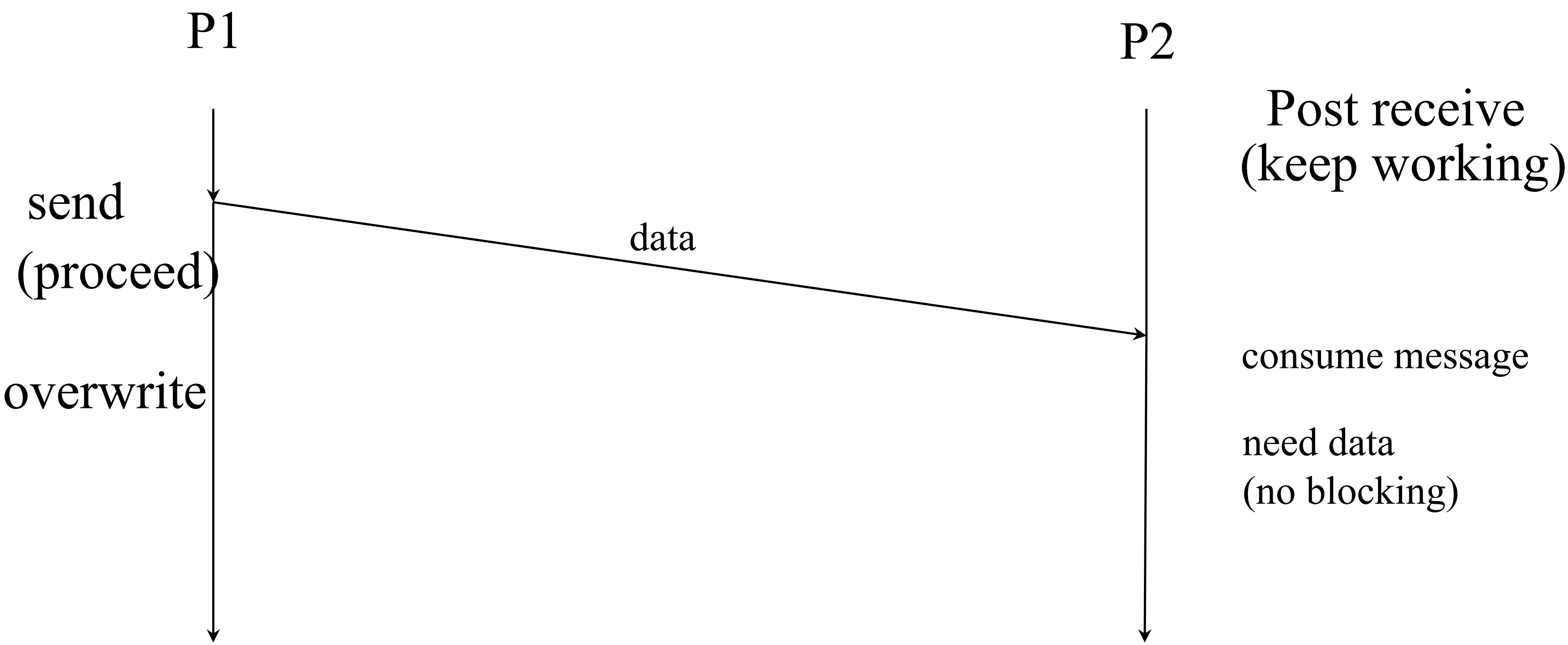
Picture of Blocking Send, Non-Blocking Receive



Picture of Non-Blocking Send, Non-Blocking Receive



Picture of Non-Blocking Send, Non-Blocking Receive



Possible Implementation of Non-Blocking Send, Blocking Receive

- Library must keep track of all channels
  - one queue and one semaphore (initialized to zero) per channel on the receiver
- On Send(channel, message)
  - copy message into send buffer (will end up on network if receiver on remote machine)
- On Receive(channel, message)
  - P(thisQueue); copy proper data into message
- On incoming message (specifies channel)
  - buffer it in the queue; V(thisQueue)

Possible Implementation of Blocking Send, Blocking Receive

- Library must keep track of all channels
  - one queue and one semaphore (initialized to zero) per channel on both ends
- On Send(channel, message)
  - copy message into send buffer (will end up on network if receiver on remote machine); **P(ack)**
- **On incoming ack (specifies channel)**
  - **V(ack)**
- On Receive(channel, message)
  - P(thisQueue); copy proper data into message; **send ack to sender**
- On incoming message (specifies channel)
  - buffer it in queue; V(thisQueue)

Tradeoffs in Message Passing

- Advantages of blocking send
  - won’t overwrite message; less buffering; cannot overwhelm receiver
- Advantages of non-blocking send
  - can continue after send (can do other work); deadlock less likely
- Advantages of blocking receive
  - know message is received, simpler app. code
- Advantages of non-blocking receive
  - can result in fewer copies (buffer posted in advance); can allow blocking sender to resume earlier

Realizing Message Passing in MPI

- Blocking/non-blocking operations
  - Send, Ssend, Isend, Recv, Irecv
  - Send and Recv are not analogous (confusing)
    - Send may block; Recv will block
    - Ssend and Recv require matching in all cases
- Collective operations
  - Barrier, Scatter, Gather, Alltoall
  - Why use these as opposed to several point-to-point messages?
    - Convenience
    - Efficiency
      - Implementation can optimize when it knows what's coming
  - Can be used over a subset of processes (not shown yet)

Programming Client/Server Applications

Outline of Client code

```
while (1)   {
  build request
  send (request, server)
  receive (reply)
  do something
}
```

Outline of Server code

```
while (1)  {
  receive (request)
  switch (request)
  case type1:
     send (client, reply1)
  case type2:
     send (client, reply2)
  etc.
}
```

Duality of Monitors/Message Passing (Lauer, H.C., Needham, R.M., "On the Duality of Operating Systems Structures", 1978)

Monitors	Message Passing
Monitor variables	Local vars on server
Entry (implicit mutex)	Blocking recv on server
Procedures in monitor	Arms of switch stmt
Procedure call	Client sends request to server; may block awaiting reply
Procedure return	Server sends result to appropriate client
Wait	Insert request on server queue
Signal	Remove & process request from server queue

Duality Example: Resource Allocation with Monitors

```
monitor ResourceAllocator
  int free = true; cond c
  acquire( ):       if (free) free = false
                    else wait(c)
  release( ):       if (empty(c)) free = true
                    else signal(c)
end ResourceAllocator
```

Resource Allocation with Message Passing

```
Client (i) {
    send request (i, ACQUIRE)
    receive reply[i]( )
    send request (i, RELEASE)
}
```

Resource Allocation with Message Passing

```
enum reqType {ACQUIRE, RELEASE)
chan request(int clientId, reqType which)
chan reply[n]( )   // one entry per client
Allocator {        // runs on server
  queue pending;  # initially empty
  int clientId;  bool free := true
  enum which {ACQUIRE, RELEASE};

// (continued on next slide)
```

Resource Allocation with Message Passing

```
  while (1) {
    receive request(clientId, which)
    switch(which) {
    ACQUIRE:
        if (free)
          free := false; send reply[clientId]
        else
          pending.insert(clientId)
    RELEASE:
        if notempty(pending)
          send reply[pending.remove()]
        else
          free := true
    }
  }
} // end of Allocator
```