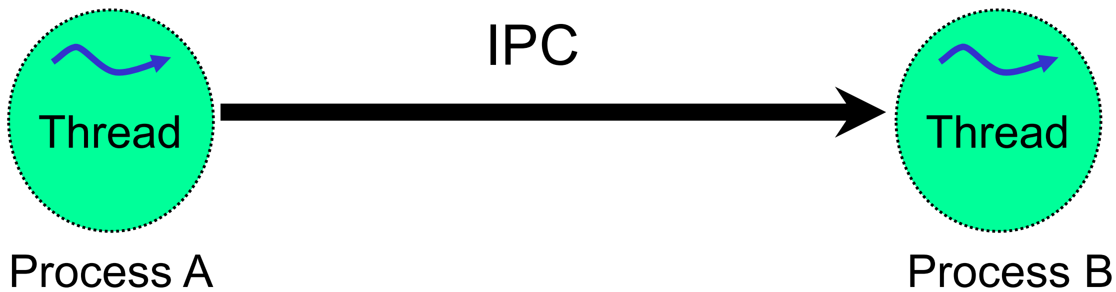
QNX
Inter-Process Communication

IPC:

- Inter-Process Communication
- two processes exchange:
  - data
  - control
  - notification of event occurrence



Introduction

QNX Neutrino supports a wide variety of IPC:

- QNX Core (API is unique to QNX or low-level)
  - includes:
    - QNX Neutrino messaging
    - QNX Neutrino pulses
    - shared memory
  - the focus of this course module
- POSIX/Unix (well known, portable API's)
  - includes:
    - signals
    - shared memory
    - pipes (requires **`pipe`** process)
    - POSIX message queues (requires **`mqueue`** or **`mq`** process)
    - TCP/IP sockets (requires **`io-pkt-*`** process)
  - the focus of the POSIX IPC course module

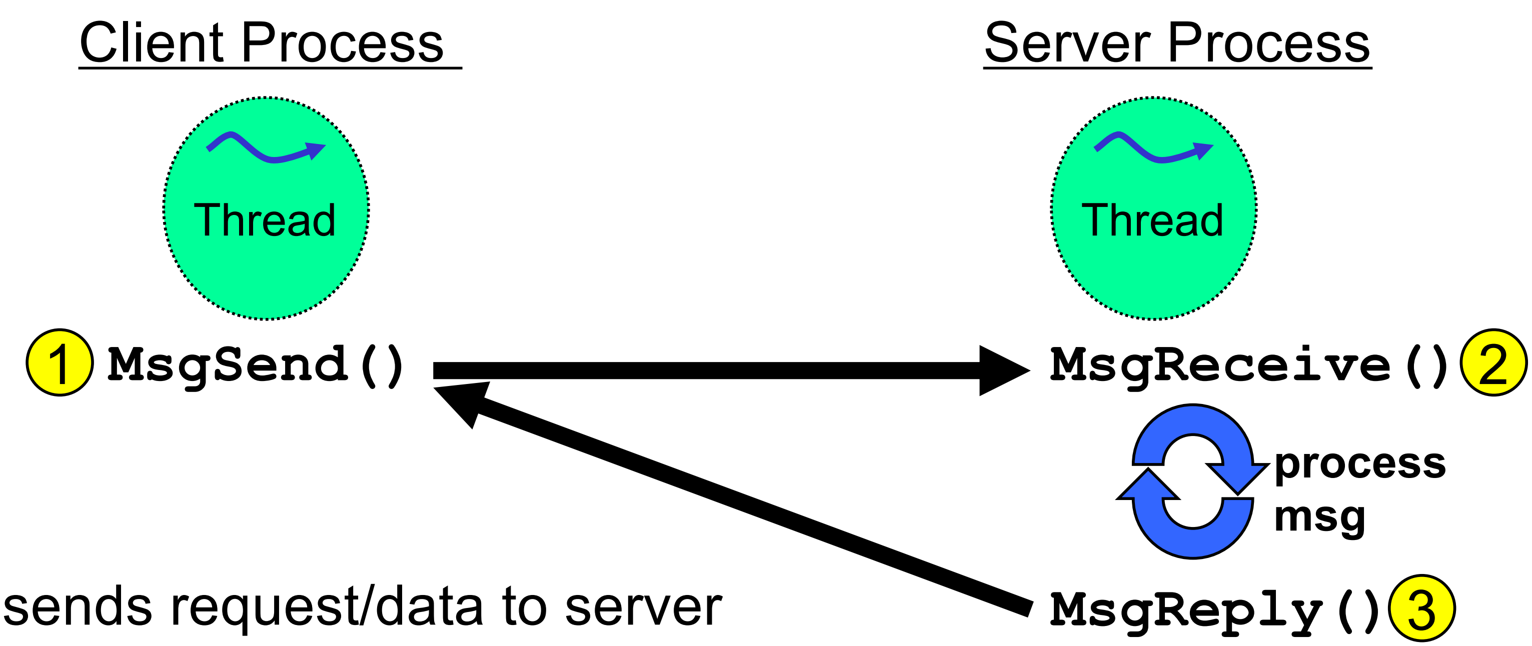
Topics:

- **Message Passing**
- **Designing a Message Passing System (1)**
- **Pulses**
- **How a Client Finds a Server**
- **Client Information Structure**
- **Server Cleanup**
- **Multi-Part Messages**
- **Designing a Message Passing System (2)**
- **Issues Related to Priorities**
- **Designing a Message Passing System (3)**
- **Event Delivery**
- **Shared Memory**
- **Conclusion**

Native QNX Neutrino Messaging

QNX Native IPC:

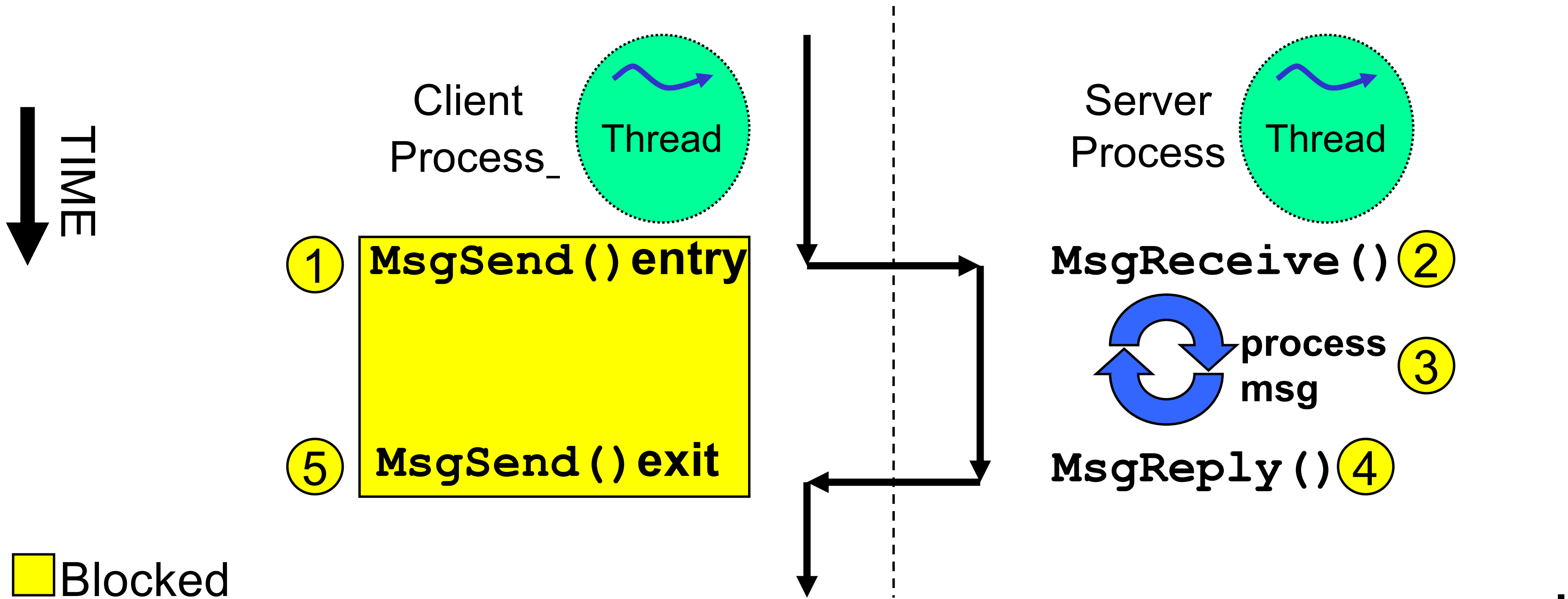
- client-server based
- bidirectional communication



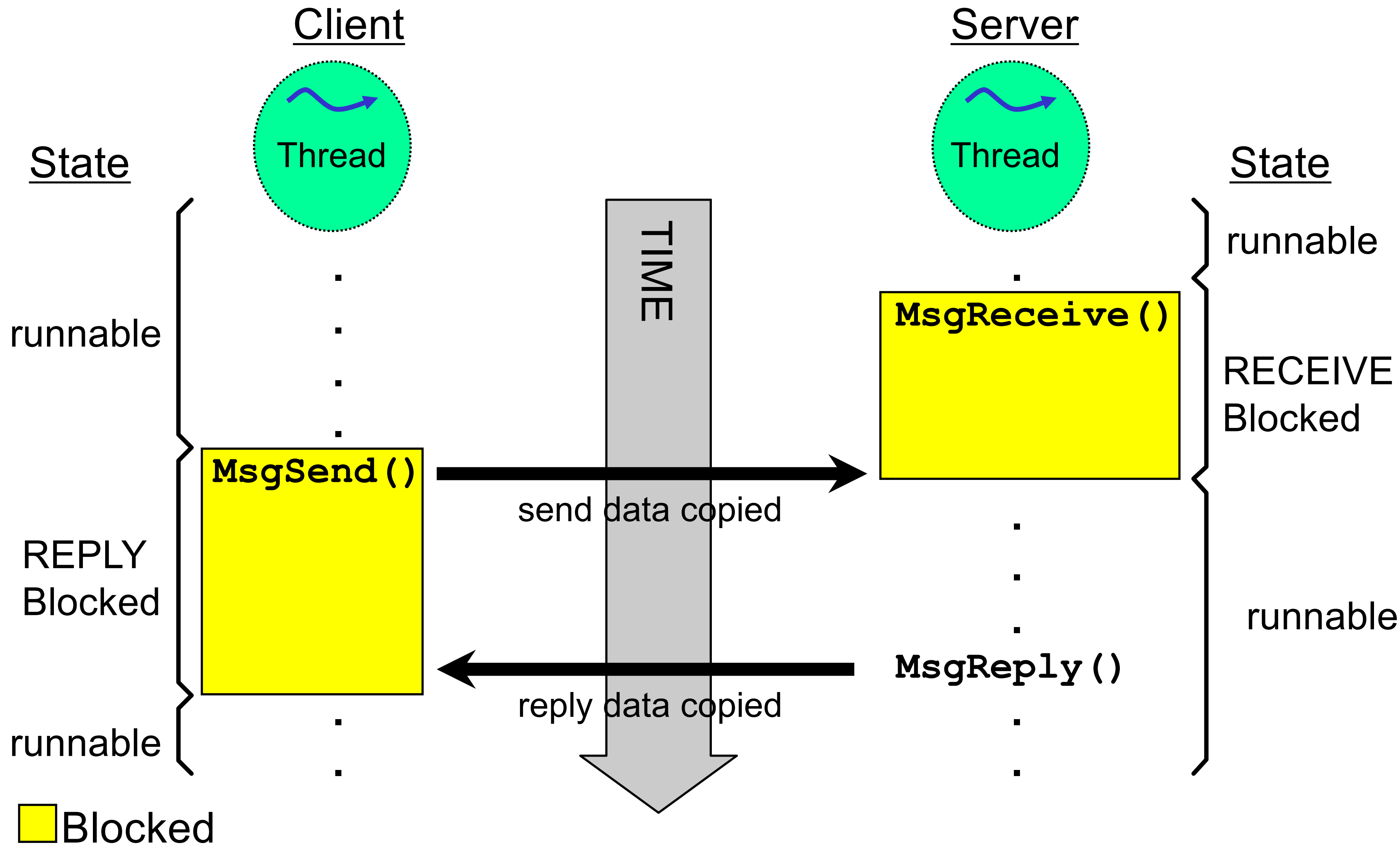
1. client sends request/data to server
2. server receives, and processes
3. server replies to client, client gets response and continues

QNX Native IPC is:

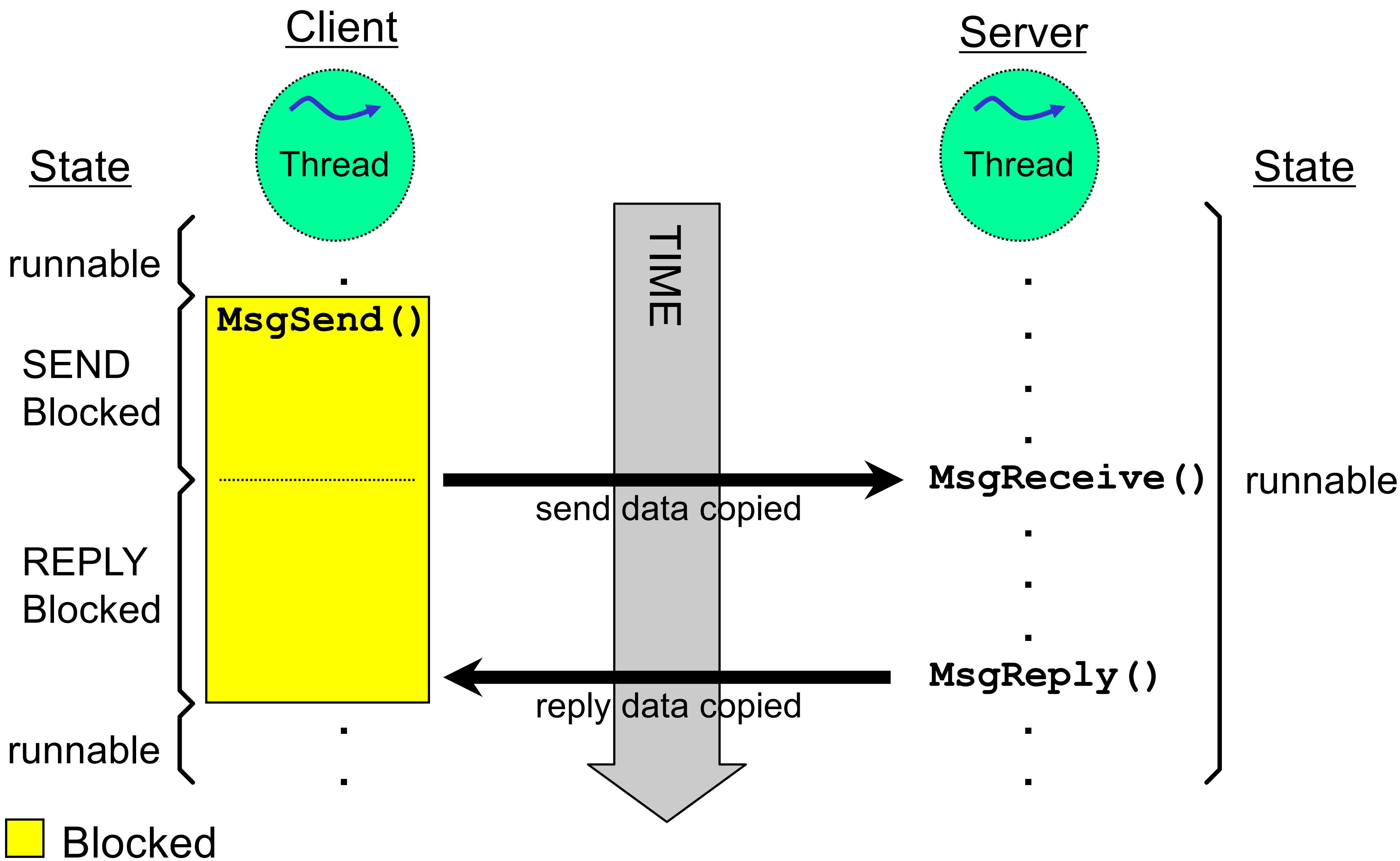
- a remote procedure call architecture
  - server thread runs at client priority
- fully synchronous
  - when idle, server is blocked waiting for message
  - client blocks until server replies



CASE 1: Send after Receive – blocked/idle server

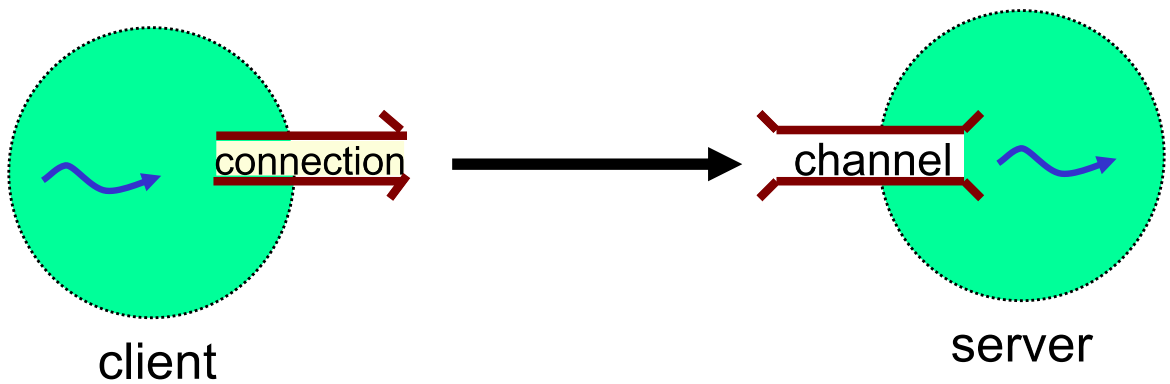


CASE 2: Send before Receive – busy server



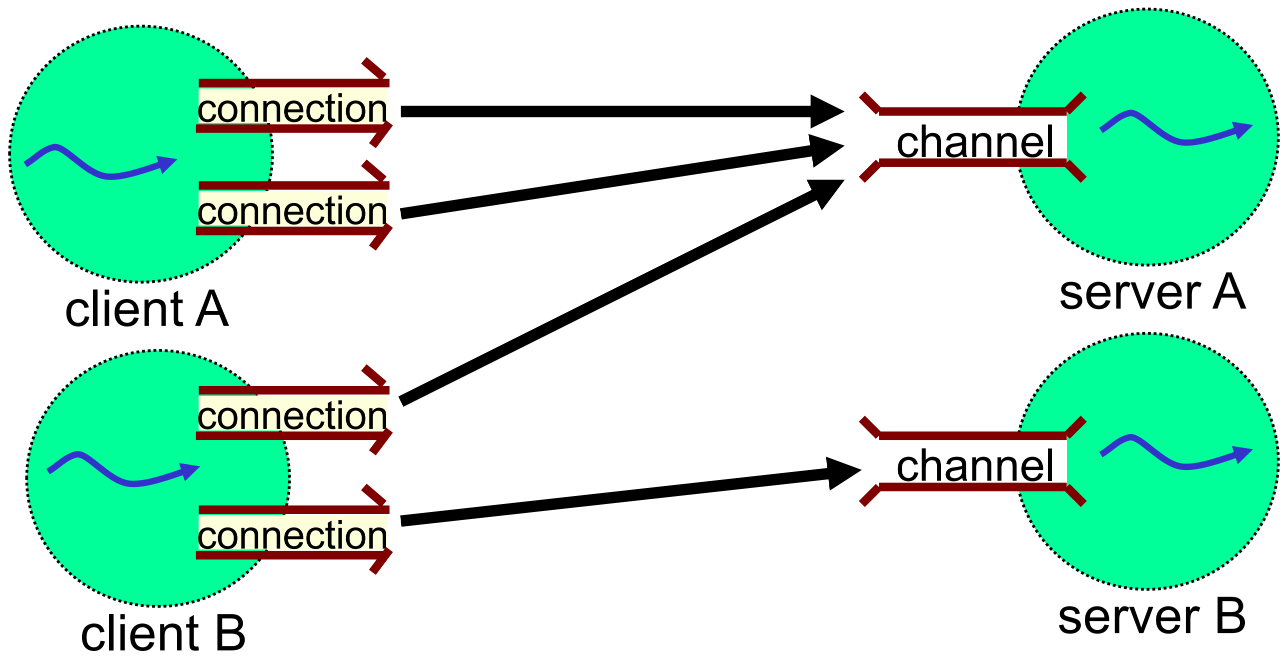
Connections and channels:

- servers receive on channels,
- clients connect to channels
- a client sends to a server’s channel using the connection



Multiple Connection and Channels:

- a client may have connections to several servers
- a server uses a single channel to receive messages from multiple clients



Server pseudo-code:

- create a channel (*ChannelCreate()*)
- wait for a message (*MsgReceive()*)
  - perform processing
  - reply (*MsgReply()*)
- go back for more

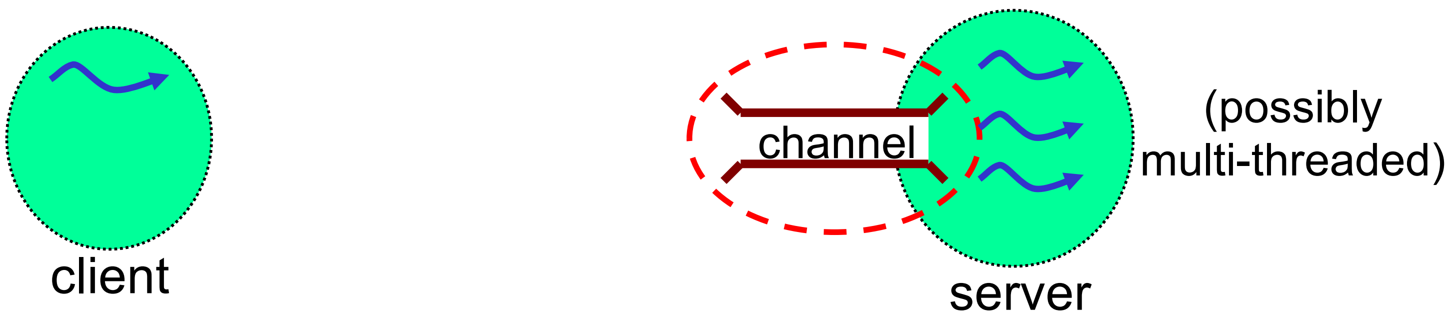
Client pseudo-code:

- attach to a channel (*ConnectAttach()*)

  ...
- send message (*MsgSend()*)
- make use of reply

The server creates a channel using:

```
chid = ChannelCreate (flags);
```



- a channel is attached to a process
- any thread in the process can receive messages from the channel if it wants to
- when a message arrives, the kernel simply picks a *MsgReceive()*ing thread to receive it
- **`flags`** is a bitfield, we will look at some of the flags on the next slide

Some of the *ChannelCreate()* flags:

`_NTO_CHF_PRIVATE`
- a channel for a process' internal use
- to create a public channel your process must have the PROCMGR_AID_PUBLIC_CHANNEL ability

`_NTO_CHF_THREAD_DEATH`
- notify on the death of any thread in the process that owns the channel

`_NTO_CHF_COID_DISCONNECT`
- notify when a connection (coid or fd) becomes invalid

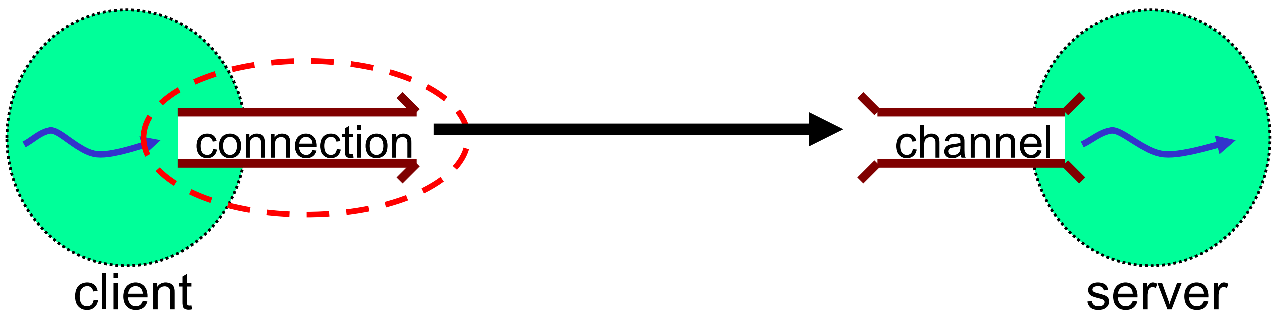
`_NTO_CHF_DISCONNECT`
- notify when a client process goes away

`_NTO_CHF_UNBLOCK`
- notify when a client thread needs to be unblocked

The last two flags require user work, and will be discussed in the Server Cleanup section

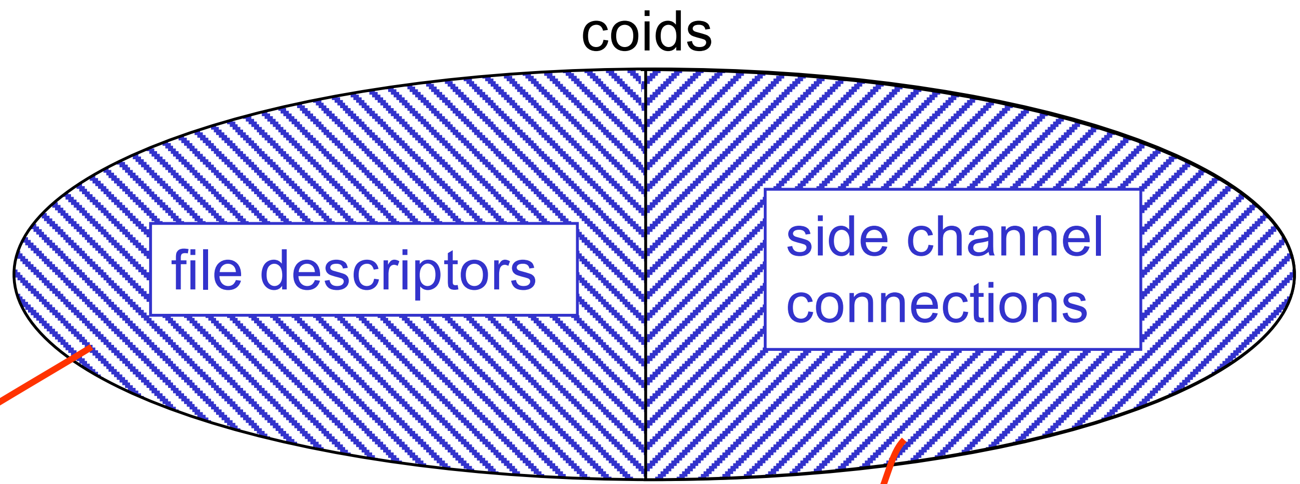
The client connects to the server's channel:

```
coid = ConnectAttach(0, pid, chid, _NTO_SIDE_CHANNEL, flags);
```



- **`pid`**, **`chid`** uniquely identify the server's channel
  - **`pid`** is the process id of the server
  - **`chid`** is the channel id
  - always pass **`_NTO_SIDE_CHANNEL`** for the 4th parameter (index) -- why? ...

Connection IDs (coids) come in two types:




```
fd = open (filename, O_RDONLY);

coid = ConnectAttach (0, pid, chid, _NTO_SIDE_CHANNEL, flags);
```

- when calling *ConnectAttach()* yourself (as opposed to a lib function calling it) you do not want your coid to be in the file descriptor range
- passing **`_NTO_SIDE_CHANNEL`** prevents it

To get a listing of executing threads/processes (using the IDE):

- open the System Information Perspective
- select a process in the Target Navigator
  - you can select multiple by holding down CTRL key, e.g. select server and client
- the Process Information View will show states of server and client

To get a listing of executing threads/processes (using the command-line), enter:

```
pidin
   pid tid name                 prio STATE       Blocked
     1   1 /procnto-smp-instr     0f READY
     1   2 /procnto-smp-instr   255r RECEIVE     1
     1   3 /procnto-smp-instr   255r RECEIVE     1
     1   6 /procnto-smp-instr    10r RECEIVE     1
     1   7 /procnto-smp-instr    10r RUNNING
     1   9 /procnto-smp-instr    10r RECEIVE     1
     2   1 sbin/tinit            10r REPLY       1
  4099   1 proc/boot/pci-bios    12r RECEIVE     1
 20489   1 sbin/pipe             10r SIGWAITINFO
 20489   2 sbin/pipe             10r RECEIVE     1
 20489   3 sbin/pipe             10r RECEIVE     1
 20489   4 sbin/pipe             10r RECEIVE     1
 20489   5 sbin/pipe             10r RECEIVE     1
 69645   1 sbin/enum-devices     10r REPLY       20489
536611   1 bin/pidin             10r REPLY       1
```

For RECEIVE blocked state, refers to the chid it's blocked on

If REPLY blocked, this is the pid of the server we're waiting for the reply from

Exercise: examine QNX message passing in a system:

- try out the IDE views:
  - Process Information
  - Connection Information

and/or,

- try out the command line tools
  - **`pidin`**
  - **`pidin fd`**
- notice the threads that are in QNX message passing states?
  - message passing is at the heart of every Neutrino system
- find a **`ksh`** or **`login`** process
  - what process is it blocked on? What process(es) are those blocked on?

The MsgSend() call (client):

```
status = MsgSend (coid, smsg, sbytes, rmsg, rbytes);
```

- **`coid`** is the connection ID
- **`smsg`** is the message to send
- **`sbytes`** is the number of bytes of **`smsg`** to send
- **`rmsg`** is a reply buffer for the reply message to be put into
- **`rbytes`** is the size of **`rmsg`**

`status` is what will be passed as the *MsgReply*()*'s **`status`** parameter

The MsgReceive() call (server):

```
rcvid = MsgReceive (chid, rmsg, rbytes, info);
```

`chid` is the channel ID

`rmsg` is a buffer in which the message is to be received into

`rbytes` is the number of bytes to receive in **`rmsg`**

`info` allows us to get additional information

`rcvid` allows us to *MsgReply*()* to the client

The MsgReply() call (server):

```
MsgReply (rcvid, status, msg, bytes);
```

`rcvid` is the receive ID returned from the server's *MsgReceive*()* call

`status` is the value for the *MsgSend*()* to return, do not use a negative value

`msg` is reply message to be given to the sender

`bytes` is the number of bytes of **`msg`** to reply with

The MsgError() call (server):

– will cause the *MsgSend*()* to return -1 with **`errno`** set to whatever is in **`error`**.

```
MsgError (rcvid, error);
```

`rcvid` is the receive ID returned from the server's *MsgReceive*()* call

`error` is the error value to be put in **`errno`** for the sender

The server:

```
#include <sys/neutrino.h>

int main(void) {
   int chid, rcvid;
   mymsg_t msg;
   …

   chid = ChannelCreate(0);

   while(1) {
      rcvid = MsgReceive(chid, &msg, sizeof(msg), NULL);

      … perform processing on message/handle client request

      MsgReply(rcvid, EOK, &reply_msg, sizeof(reply_msg) );
   }
}
```

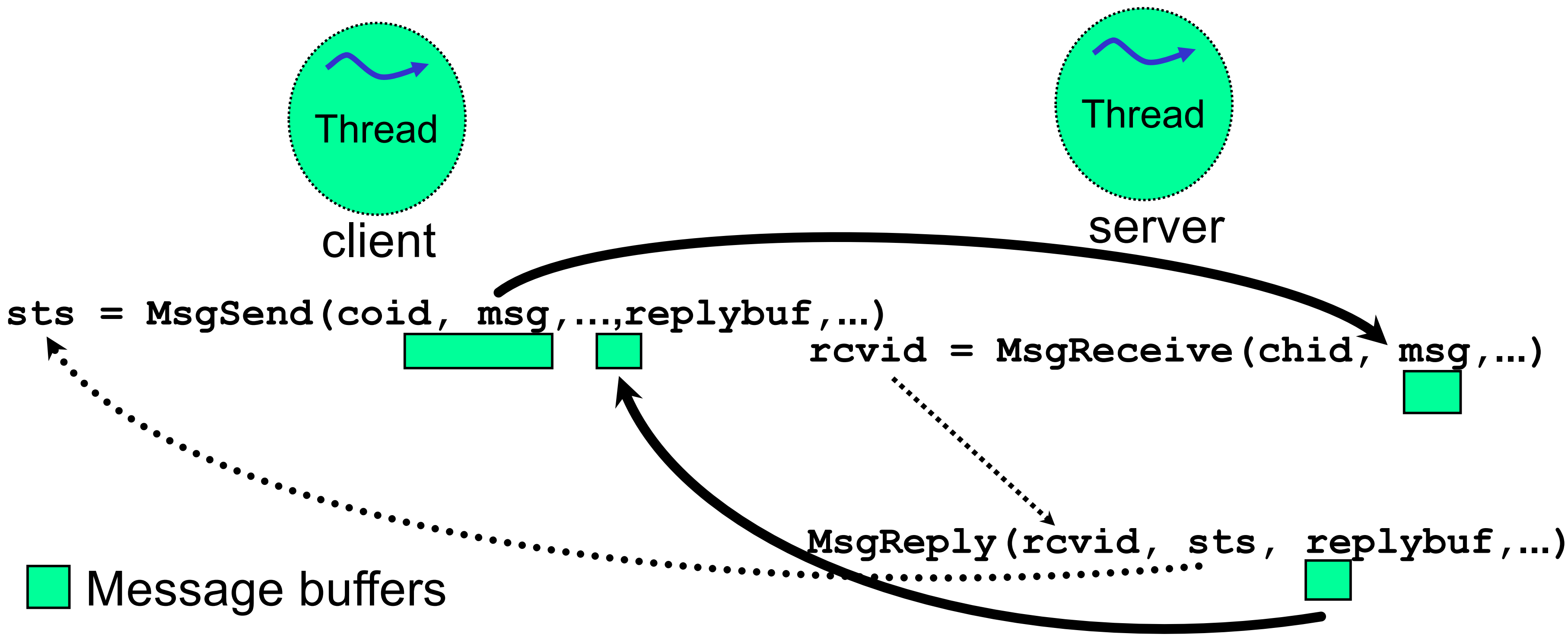
The client:

```
#include <sys/neutrino.h>
#include <sys/netmgr.h>

int main(void)
{
   int coid;    //Connection ID to server
   mymsg_t  outgoing_msg;
   int server_pid, server_chid, incoming_msg;
   …
   coid = ConnectAttach(0, server_pid, server_chid,
   _NTO_SIDE_CHANNEL, 0);
   …
   MsgSend(coid, &outgoing_msg, sizeof(outgoing_msg),
                          &incoming_msg, sizeof(incoming_msg) );
   …
}
```

Messaging - The message data

Message data is always copied:

– the kernel does not pass pointers

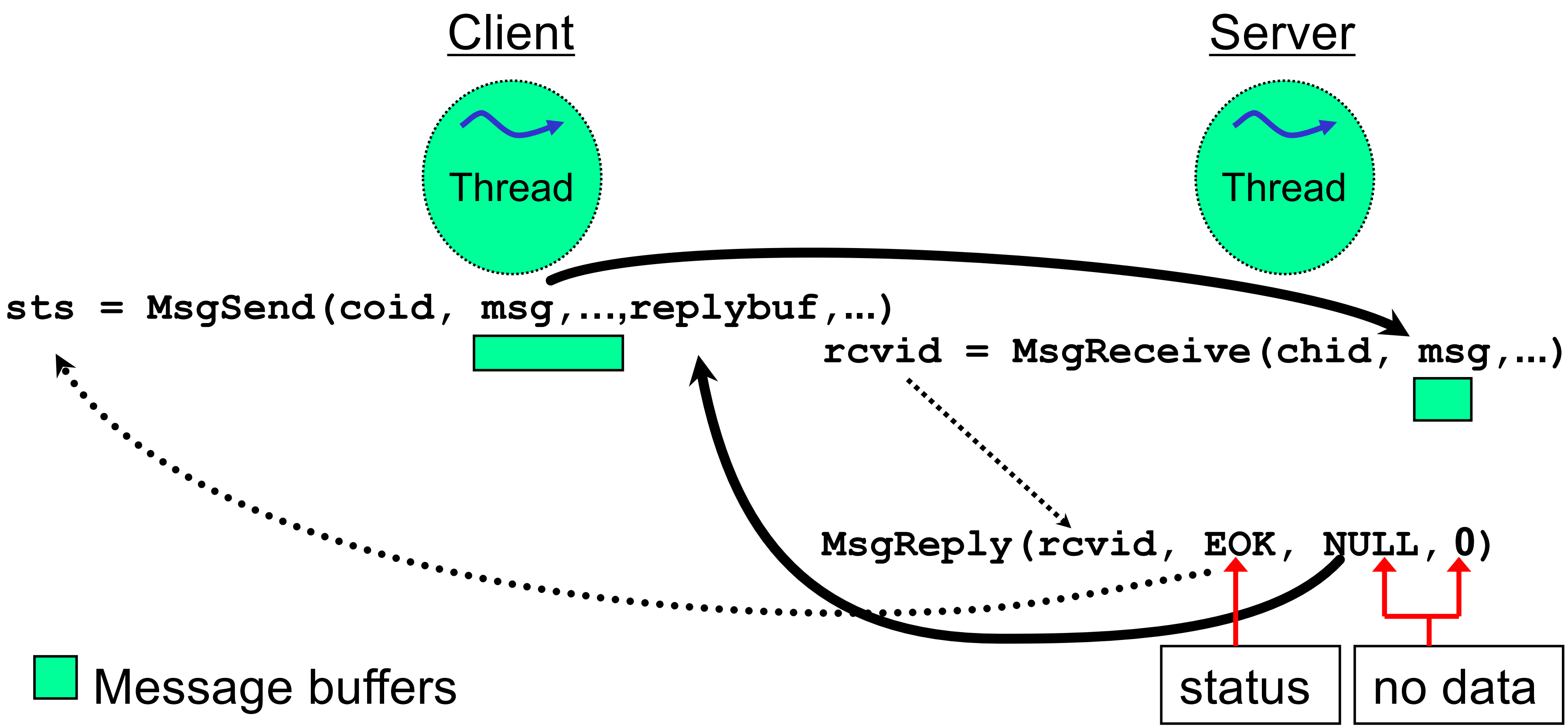


– the number of bytes actually transferred is the smaller of the client buffer size and the server buffer size

Messaging - The message data

A server can reply with no data:

- this can be used to unblock the client, in cases where you only need to give a status/acknowledgement back to *MsgSend()*
- client knows the status of its request



Exercise: basic Send/Receive/Reply

- in your `ipc` project are two files, `server.c` and `client.c`
- the server is a "checksum server", it works as follows:
  - the client sends a string to the server
  - the server receives the message
  - the server calculates a checksum on the string
  - the server replies back to the client with the checksum
- both client and server are incomplete, trace through the code looking for comments indicating where code should be added
- build client/server
- run the server, write down, or 'copy' its **pid** and **chid**
- run the client, using the **pid**, **chid**, and some text of your choice as command-line arguments
- observe the results

continued...

Exercise: basic Send/Receive/Reply (continued):

- some questions to consider:
  1. what states did the client and server transition through?
     - consider stepping the programs in the debugger and the Process Information View
  2. did the client ever become SEND blocked?
  3. if you were to remove the server's *MsgReply()*, and re-run client and server, what would be the result? Why?
  4. if the server's *MsgReceive()* returns a failure, should the program exit?
     - what are some reasons that could cause *MsgReceive()* to return -1? (check documentation for *MsgReceive()*)
  5. under normal circumstances, the client prints out:

     "`MsgSend return status: <some value>`"
     - where did that value come from?

Topics:

- Message Passing
- → Designing a Message Passing System (1)
- Pulses
- How a Client Finds a Server
- Client Information Structure
- Server Cleanup
- Multi-Part Messages
- Designing a Message Passing System (2)
- Issues Related to Priorities
- Designing a Message Passing System (3)
- Event Delivery
- Shared Memory
- Conclusion

How do you design a message passing interface?

- define message types and structures in a header file
  - both client and server will include the common header file
- start all messages with a message type
- have a structure matching each message type
  - if messages are related or they use a common structure, consider using message types & subtypes
- define matching reply structures, if appropriate
- avoid overlapping message types for different types of servers

Avoid overlapping with QNX system message range:

- these types of messages are generated by QNX system library routines, e.g. *read()*
- all QNX messages start with:

  `uint16_t type`

- which will be in the range:

  `0` to **`_IO_MAX`** (511)

- using a value greater than **`_IO_MAX`** is always safe

On the server side

- branch based on message type, e.g.:

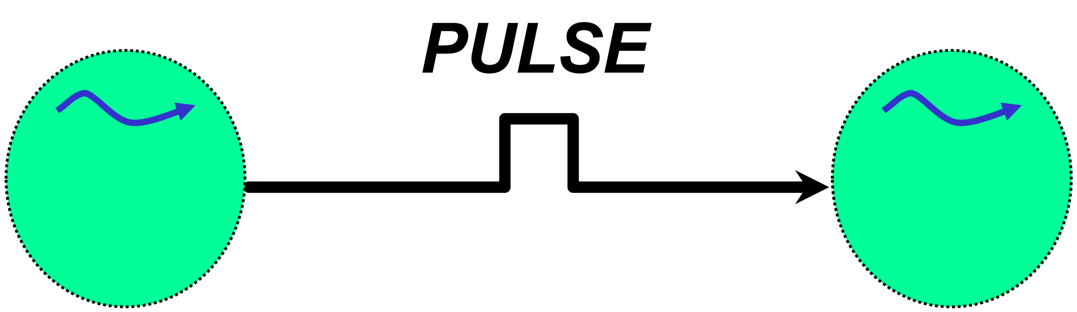
```
while(1) {
  rcvid = MsgReceive( chid, &msg, sizeof(msg),
  NULL );
  switch( msg.type ) {
      case MSG_TYPE_1:
        handle_msg_type_1(rcvid, &msg);
        break;
      case MSG_TYPE_2:
      …
   }
}
```

Topics:

- Message Passing
- Designing a Message Passing System (1)
- Pulses
- How a Client Finds a Server
- Client Information Structure
- Server Cleanup
- Multi-Part Messages
- Designing a Message Passing System (2)
- Issues Related to Priorities
- Designing a Message Passing System (3)
- Event Delivery
- Shared Memory
- Conclusion

Pulses:

- non-blocking for the sender
- fixed-size payload
  - 32 or 64 bit value
    - for intra-process use pulses can carry a pointer value, and this would be a 64-bit value in 64-bit processes
  - 8 bit code (-128 to 127)
    - negative values reserved for system use
    - 7-bits available
- unidirectional (no reply)
- fast and inexpensive



Pulses are sent as follows:

```
MsgSendPulse (coid, priority, code, value);
```

8-bits
32-bits

- **`code`** is usually used to mean "pulse type"
  - valid range is **`_PULSE_CODE_MINAVAIL`** (0) to **`_PULSE_CODE_MAXAVAIL (127)`**
- to send a pointer use *MsgSendPulsePtr()* instead
- **`priority`** indicates the priority the receiving thread should run at
  - QNX uses a priority inheritance scheme to minimize priority inversion problems
  - delivery order is based on priority
  - -1 can be used for the priority of the calling thread
- to send a pulse across process boundaries, the sender must have appropriate privilege

Receiving Pulses

Pulses are received just like messages, with a *MsgReceive*()* call:

- a server can determine whether it has received a pulse vs. a message by the return value from *MsgReceive()*
  - the return value will be >`0` if a **message** was received.
    - this value will be the **`rcvid`**, which will be needed to *MsgReply()*
  - the return value will be == `0` if a **pulse** was received
    - you can not *MsgReply()* to pulses
- the pulse data will be written to the receive buffer
  - the receive buffer must be large enough to hold the pulse structure, if not:
    - the return value will be == -1, and **`errno`** will be **`EFAULT`**
    - the pulse will be lost

Example:

```
typedef union {
    struct _pulse   pulse;
    // other message types you will receive
} myMessage_t;


…
myMessage_t   msg;

while (1) {
    rcvid = MsgReceive (chid, &msg, sizeof(msg), NULL);
    if (rcvid == 0) {
        // it's a pulse, look in msg.pulse… for data
    } else {
        // it's a regular message
    }
}
…
```

Pulse Structure

When received, the pulse structure has at least the following members:

```
struct _pulse {
  signed char         code;
  union sigval        value;
  int                 scoid;
};
```

8-bit code

The value member is actually a union:

```
union sigval {
  int sival_int;
  void* sival_ptr;
};
```

The server will want to determine the reason for this pulse, by checking the pulse code

```
rcvid = MsgReceive (chid, &msg, sizeof(msg), NULL);
if (rcvid == 0) {
    // it’s a pulse, look in msg.pulse… for code
    switch (msg.pulse.code) {
    case    MY_PULSE_CODE1:
            // do what’s needed
            ...
            break;
    case    MY_PULSE_CODE2:
            // do what's needed
            ...
            break;
```

Pulse Exercise:

- in your **`ipc`** project
  - copy the checksum **`server.c`** from last exercise to **`pulse_server.c`**
  - uncomment the line in the **`Makefile`** to build the pulse server
- extend the server so that it can now handle pulses as well as checksum request messages
- whenever the server receives a pulse, it should print that it got a pulse, and print the pulse data (code and value)
- **`pulse_client.c`** is a copy of **`client.c`** that sends a pulse as well as a message
- run the server, write down, or 'copy' the **pid** and **chid**
- run the pulse client, using the **pid**, **chid**:
  - verify the exchange of messages and delivery of pulses

Topics:

- Message Passing
- Designing a Message Passing System (1)
- Pulses
- How a Client Finds a Server
- Client Information Structure
- Server Cleanup
- Multi-Part Messages
- Designing a Message Passing System (2)
- Issues Related to Priorities
- Designing a Message Passing System (3)
- Event Delivery
- Shared Memory
- Conclusion

How does the client find the server?

- for a client to send to a server it needs a connection ID (coid)
  - i.e. `MsgSend(coid, &msg, ….)`
- as we've seen, to get a coid, a client can do **`ConnectAttach()`**
  - i.e. `coid = ConnectAttach(0, pid, chid,…);`
- the problem is, how does the client know what the server's **`pid`** and **`chid`** are?

continued...

How does the client find the server (cont.)?

- our exercises had the server print out its **`pid`**/**`chid`**, and the client took them as command-line arguments
  - this doesn't work well as a general solution
- there are two other methods, depending on whether the server is:
  - a resource manager
  - a simple *MsgReceive()* loop
- both methods make use of the pathname space
  - server puts a name into the path name space
  - both client and server must have an understanding of what that name will be
  - client does an 'open' on the name, and gets a **`coid`**

How does the client find the server? - Resource manager

If the receiver is a resource manager:

- the resource manager attaches a name into the namespace:

```
resmgr_attach( ..., "/dev/sound", ...  );
```

- the client does:

```
fd = open( "/dev/sound", ... );
```

- then it can make use of the **fd** (recall that **fd**'s are a particular type of **coid**)

```
write( fd, ... );   // sends some data
read( fd, ... );    // gets some data
```

OR

```
MsgSend( fd, ... ); // send data, get data back
```

Finding the server - name_attach()/name_open()

If the server is a simple *MsgReceive()* loop

– use *name_attach()* and *name_open()*:

The server does:

```
name_attach_t *attach;
attach = name_attach( NULL, "myname", 0 );
...
rcvid = MsgReceive( attach->chid, &msg, sizeof(msg),
                    NULL );
...
name_detach( attach, 0 );
```

The client does:

```
coid = name_open( "myname", 0 );
...
MsgSend( coid, &msg, sizeof(msg), NULL, 0 );
...
name_close( coid );
```

Channel flags

name_attach() creates the channel for you:

- internally it does a *ChannelCreate()*
- when doing so, it turns on several channel flags
- the channel flags request that the kernel send pulses to provide notification of various events
- your code should be aware that it will get these pulses, and handle them appropriately

ChannelCreate flags

ChannelCreate() flags that *name_attach()* sets:

`_NTO_CHF_DISCONNECT`

- requests that the kernel deliver the server a pulse when a client goes away
- pulse will have code _PULSE_CODE_DISCONNECT

`_NTO_CHF_COID_DISCONNECT`

- requests that the kernel deliver the client a pulse when a server goes away
  - it is possible for a client to have a channel, as we'll see later
- pulse will have code _PULSE_CODE_COIDDEATH
- value will be the coid/fd that has become invalid

`_NTO_CHF_UNBLOCK`

- requests that the kernel deliver a pulse if a REPLY blocked client wants to unblock
- pulse will have code _PULSE_CODE_UNBLOCK

Example of a server receiving kernel pulses:

```
rcvid = MsgReceive(attach->chid, &msg, sizeof(msg), NULL);
if(rcvid == 0) {  //did we receive a pulse?
  switch(msg.pulse.code) {    //what kind of pulse is it?
      case _PULSE_CODE_DISCONNECT: //client disconnected
      …
      break;
      case _PULSE_CODE_UNBLOCK:  //client wants to unblock
      …
      break;
      case … //others
      …
```

Exercise: how a client finds the server

- again, you will continue extending the checksum server and client files in your **ipc** project
  - copy the **pulse_server.c** and **pulse_client.c** to **name_lookup_server.c** and **name_lookup_client.c**
  - update the **Makefile** to build them
- previously the client required command-line arguments identifying the nd, pid, chid of the server
  - remove this requirement
  - modify the server so that it puts a name into the pathname space
  - modify the client to find the server by name
  - the code in the client to process the command-line arguments for the server's pid and chid can be removed
- since you will be switching your server to use *name_attach()*, and *name_attach()* creates a channel with several channel flags turned on, your server should expect to receive kernel pulses

Topics:

- Message Passing
- Designing a Message Passing System (1)
- Pulses
- How a Client Finds a Server
- → Client Information Structure
- Server Cleanup
- Multi-Part Messages
- Designing a Message Passing System (2)
- Issues Related to Priorities
- Designing a Message Passing System (3)
- Event Delivery
- Shared Memory
- Conclusion

Message Information - Getting the information

To get information about the client:

After receiving a message:

– info will be stored in this struct:

```
struct _msg_info info;
```

– can get it during the MsgReceive():

```
rcvid = MsgReceive (chid, rmsg, rbytes, &info);
```

– or later, using:

```
MsgInfo (rcvid, &info);
```

☞this will not be updated if you receive a pulse

- it will contain old (garbage) data

Message Information - struct _msg_info

The `_msg_info` structure contains at least:

`pid_t`	`pid;`	sender's process ID
`int`	`tid;`	sender's thread ID
`int`	`chid;`	channel ID
`int`	`scoid;`	server connection ID
`int`	`coid;`	sender's connection ID
`short`	`priority;`	sender's priority
`short`	`flags;`	extra info
`size_t`	`msglen;`	msg lengths
`size_t`	`srcmsglen;`	msg lengths
`size_t`	`dstmsglen;`	msg lengths

(see next slide…)

Message size info:

```
                     info.srcmsglen           info.dstmsglen
                                 ‖                 //
status = MsgSend(coid,omsg,100,imsg,300)
```

```
rcvid = MsgReceive(chid,rcvmsg,200,&info)

                 info.msglen = 100
```

```
MsgReply(rcvid, status, replymsg,150)
```

– the number of bytes copied (info.msglen) will be the smaller of what the client sent with *MsgSend()* and what the server received with *MsgReceive()*. Similarly for the reply.

Some uses for the `_msg_info` information:

- `scoid` serves as a "client ID"
  - can be used as an index to access data structures on the server that contain information about the client
- client authentication
  - e.g. only certain client process are allowed to talk to this server
  - *ConnectClientInfo()* can be used to get further information (e.g. user id) about the client, based on the `scoid`
- message data copied verification
  - make sure data promised by (possibly untrusted) client headers has actually been copied
- reply space checking
  - how much data can I push back to the client?
- debugging and logging
  - the server may create usage logs or debug logs, and having the pid and tid logged may be useful

Topics:

- Message Passing
- Designing a Message Passing System (1)
- Pulses
- How a Client Finds a Server
- Client Information Structure
- Server Cleanup
  - Disconnect
  - Unblock
- Multi-Part Messages
- Designing a Message Passing System (2)
- Issues Related to Priorities
- Designing a Message Passing System (3)
- Event Delivery
- …

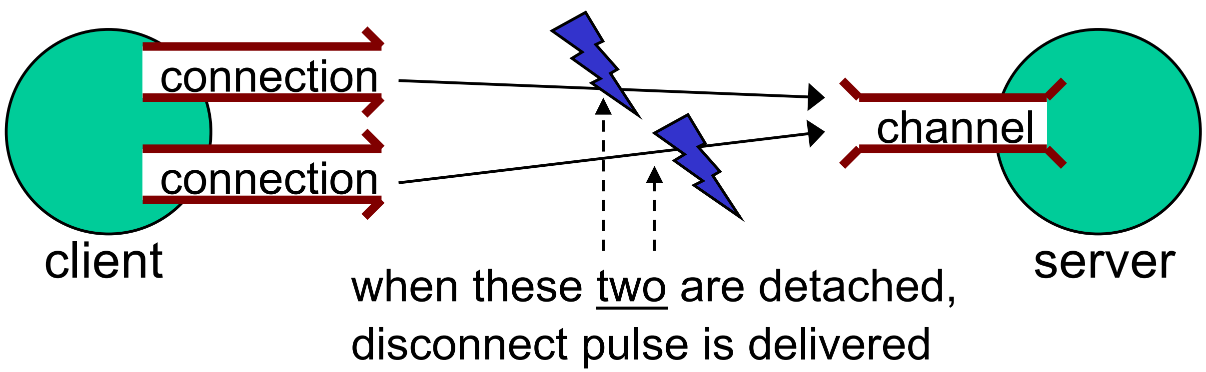
Maintaining per-client information

A server may need to maintain some information for every client process that is connected to it (per-client information):

- e.g. client status, requests pending/ongoing
- this type of information must persist as long as the client is connected to the server
- needs to be cleaned up (freed) when client disconnects
- this becomes important for event delivery, as we'll see later

The disconnect flag `_NTO_CHF_DISCONNECT`:

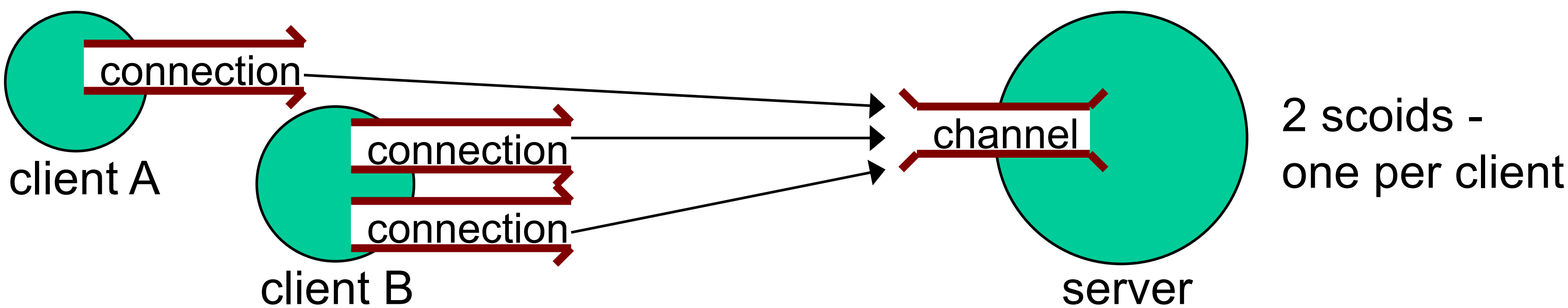
- set when channel is created
- requests that the kernel deliver the server a pulse when:
  - all connections from a particular client are detached, including:
    - process terminating
    - calling *ConnectDetach()* for all attaches



- the pulse **`code`** will be **`_PULSE_CODE_DISCONNECT`**

The scoid:

- Server Connection ID
- how server identifies a client process
  - can't use pid as client identifier, since pid's could be the same if messaging between network nodes
- a new scoid is automatically created when a new client connects
- if `_NTO_CHF_DISCONNECT` flag was specified during channel creation, scoid's have to be freed manually
- disconnect pulse means client has disconnected, you must:
  - clean up per-client information
  - do `ConnectDetach(pulse.scoid)` to free the scoid



Example cleanup for client disconnect:

```
attach = name_attach(NULL, "my_name", 0);
while (1) {
  rcvid = MsgReceive(attach->chid, &msg, sizeof(msg),NULL);
  if (rcvid == 0) { /* Pulse received */
  switch (msg.pulse.code) {
    case _PULSE_CODE_DISCONNECT:
      …//code to clean up per-client info
      ConnectDetach(msg.pulse.scoid); /* free scoid */
      break;
    …
}
```

Demo: cleanup upon client disconnect

- up to this point, our exercises have not freed any scoid's
- run disconnect_server.c and disconnect_client.c in your **`ipc`** project
- disconnect_server is the checksum server from previous exercises, except that it:
  - prints out the scoid for every client connection
  - maintains per-client info for each connected client process in a linked list
- run the disconnect_client several times, to cause several connections and disconnections to/from the server
  - notice that the scoid's keep increasing each time a client is run?
  - notice that the server never removes the per-client info from the list?

continued...

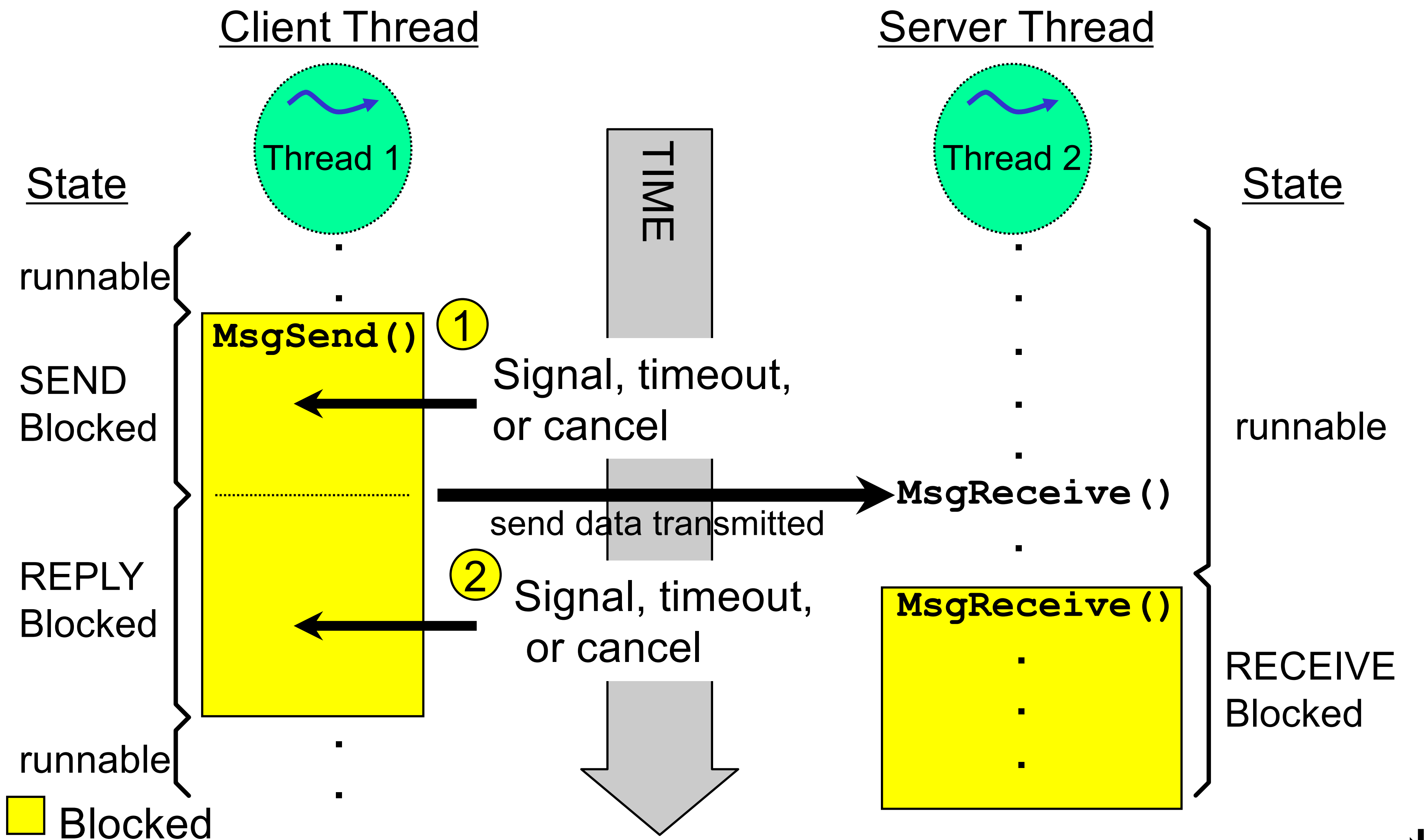
Demo: cleanup upon client disconnect (continued)

- uncomment the code for the server, so that it cleans up the **`scoid`** and the per-client info every time a client disconnects/terminates
  - *remove_client_from_list(…,scoid);* cleans up the per-client info
  - *ConnectDetach(….scoid);* cleans up the scoid
- try adding a *sleep()* before exit in the client
- rebuild and rerun
  - notice that **`scoid`** is being reused?
  - notice that the client is removed from the list when it terminates?

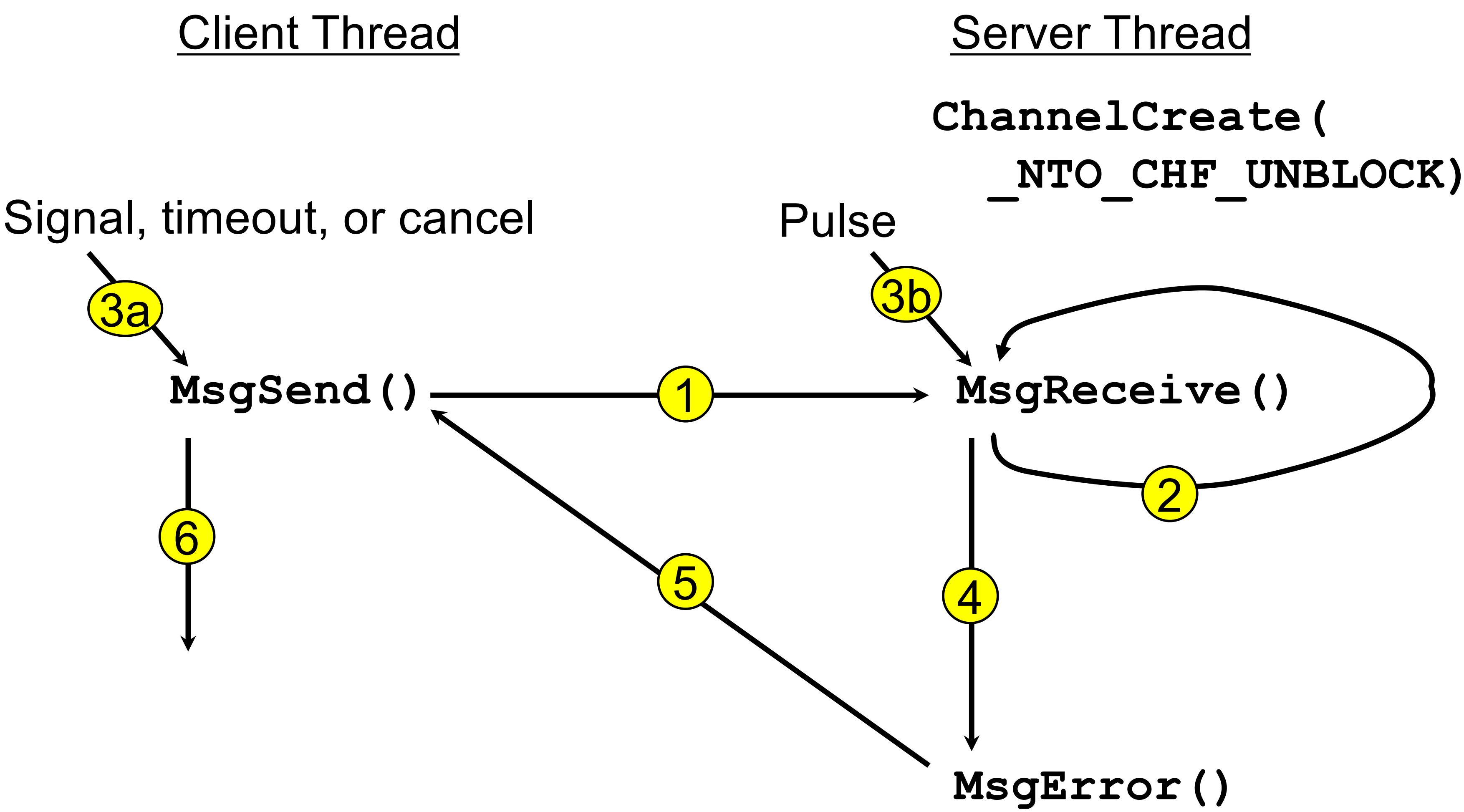
Topics:

- **Message Passing**
- **Designing a Message Passing System (1)**
- **Pulses**
- **How a Client Finds a Server**
- **Client Information Structure**
- **Server Cleanup**
  - **Disconnect**
  - **Unblock**
- **Multi-Part Messages**
- **Designing a Message Passing System (2)**
- **Issues Related to Priorities**
- **Designing a Message Passing System (3)**
- **Event Delivery**
- **…**

Client unblock problems:



The unblock flag illustrated:



See the notes for the details.

The `_NTO_CHF_UNBLOCK` flag:

- tells the kernel to deliver a pulse to the server when a REPLY blocked client thread gets a:
  - signal
  - thread cancellation
  - kernel timeout
- the **`code`** member of the pulse will be **`_PULSE_CODE_UNBLOCK`**
- the **`value.sival_int`** member will contain the rcvid that the corresponding *MsgReceive*()* returned
  - this allows the server to clean up any resources that may have been allocated for the client, and abort any operations, since the client is no longer interested in waiting for the result
- the server **MUST** *MsgReply*()* or *MsgError()* to the client, otherwise the client will remain blocked forever
  - usual choice is *MsgError(rcvid, -1)*
    - -1 is a special value that tells the kernel “use the right error”

Unblock Notification – Why?

Why does QNX do unblock notification, with the client signal delayed?

- there are several reasons:
  - we do in servers what a traditional Unix system does in the kernel
    - some operations must (according to POSIX) be atomic over signals, therefore our servers must be able to hold off signals the way a Unix kernel could
  - a server may be doing work on behalf of a client that will never get the data, we want to abort that work
    - e.g. a large read() from a filesystem, if client is interrupted/killed after a few K have been read, and won't see the data, why copy Megs more from the hardware?
  - it may be impossible to resolve the re-do/don't redo choice on the client side if the call is interrupted by a signal
    - e.g. a "debit $1000" message, if interrupted by signal… resend or not? If SEND blocked and not, no debit, if REPLY blocked and resend, debits $2000 instead of $1000. Neither is acceptable.

Demo: handling client unblock pulses

- run **`unblock_server`** and **`unblock_client`** in your **`ipc`** project
- the server will leave the client REPLY blocked
- send **`unblock_client`** a SIGTERM signal:
  - from the IDE’s Target Navigator, or
  - from the command-line, e.g. **`kill <pid>`**
- since the server has the **`_NTO_CHF_UNBLOCK`** channel flag set, the client will stay blocked, in spite of the signal
- the server will keep the client blocked for 20 seconds, then do the reply to unblock it
- the Signal Information view can be used to show pending signals
- examine the code

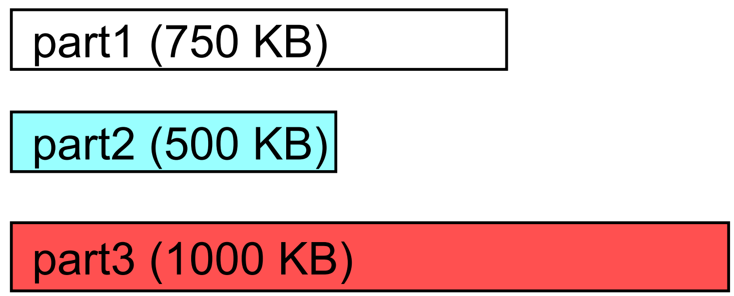
Topics:

- Message Passing
- Designing a Message Passing System (1)
- Pulses
- How a Client Finds a Server
- Client Information Structure
- Server Cleanup
- → Multi-Part Messages
- Designing a Message Passing System (2)
- Issues Related to Priorities
- Designing a Message Passing System (3)
- Event Delivery
- Shared Memory
- Conclusion

Messaging - Using IOVs

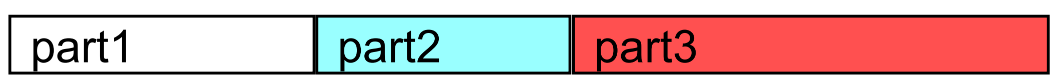
Multi-part messaging :

– what if you wanted to transfer 3 large message parts?



– you could *malloc()* enough space to hold the complete message (750 + 500 +1000 KB = 2.25 MB)

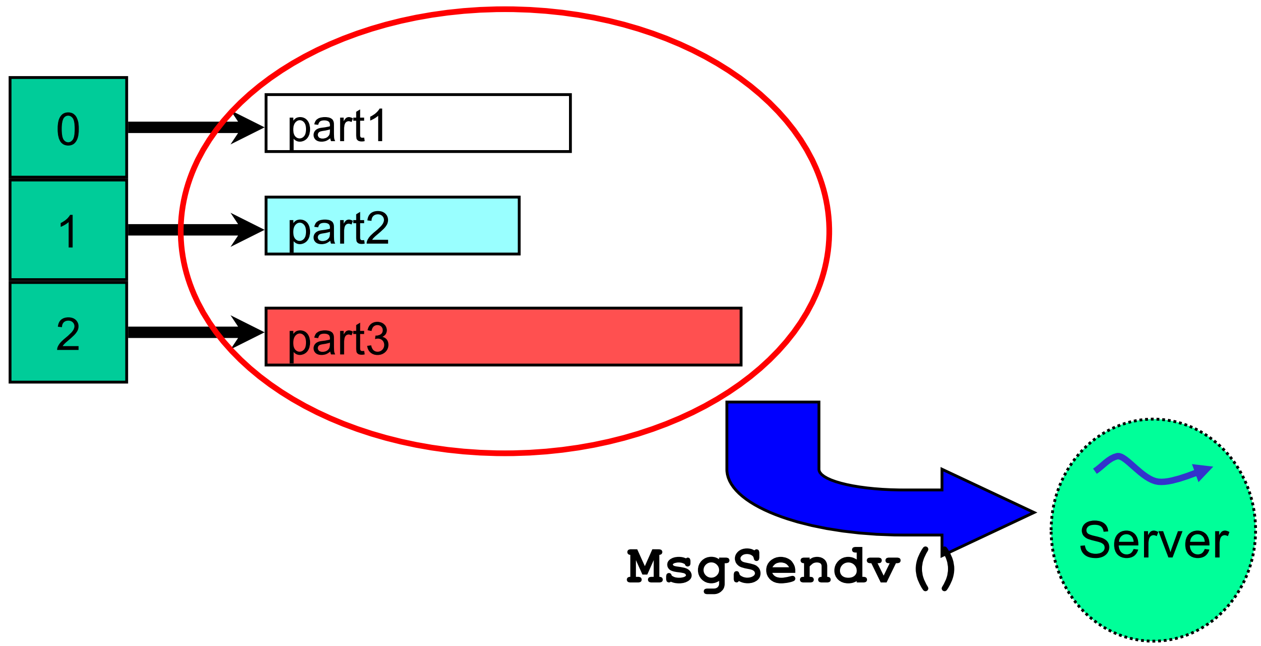
– do three *memcpy()*s to produce one big message



continued...

Multi-part messaging (cont.):

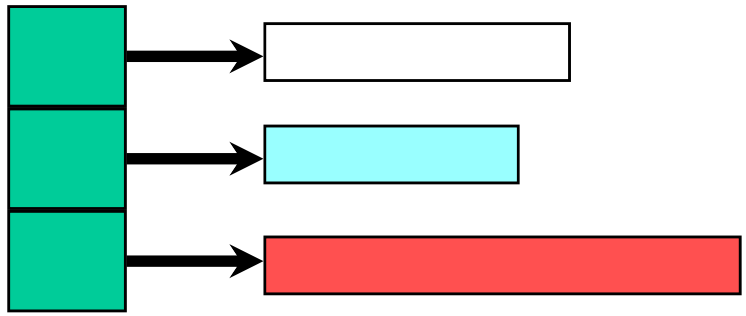
- setting up a 3-part IOV, with pointers to the data, and using *MsgSendv()* is much more efficient
  - avoids the *malloc()* and the *memcpy()*



- very useful for scatter/gather situations

IOVs are Input/Output Vectors:

- array of pointers to buffers



- uses:
  - avoiding copies when assembling messages containing multiple parts
  - variable length messages
    - server doesn't know the size of the message that the client will send
    - *write()* messages to the filesystem driver/server use IOVs

Instead of giving the kernel the address of one buffer using *MsgSend()* ...

```
MsgSend (int coid, void *smsg, int sbytes,
                void *rmsg, int rbytes);
```

one buffer

... give the kernel an array of pointers to buffers using *MsgSendv()*:

```
MsgSendv (int coid, iov_t *siov, int sparts,
                iov_t *riov, int rparts);
```

array of pointers to multiple buffers

or a combination:

```
MsgSendvs( int coid, iov_t *siov, int sparts, void *rmsg, int rbytes );
MsgSendsv( int coid, void *smsg, int sbytes, iov_t *riov, int rparts );
```

What does an `iov_t` look like?

```
typedef struct {
    void    *iov_base;
    size_t iov_len;
} iov_t;
```

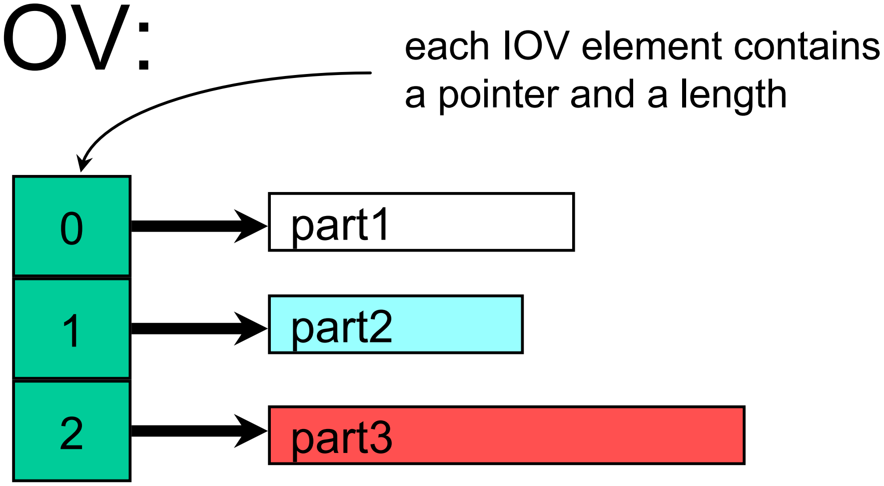
Most useful as an array:

```
iov_t      iovs [3];
```

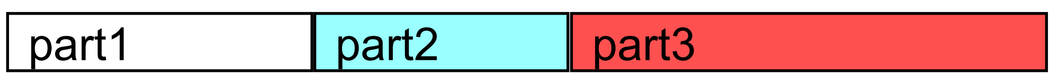
– make the number of elements >= the desired number of message parts

Messaging - Using IOVs

Using an IOV:




```
SETIOV (&iovs [0], &part1, sizeof (part1));
SETIOV (&iovs [1], &part2, sizeof (part2));
SETIOV (&iovs [2], &part3, sizeof (part3));
```



When sent or received, these parts are considered as one contiguous sequence of bytes. This is ideal for scatter/gather buffers & caches...

☞ IOVs are used in the Messaging functions that contain a “v” near the end of their name: (MsgReadv/MsgReceivev/MsgReplyv/MsgSendsv/MsgSendv/MsgSendvs/MsgWritev)

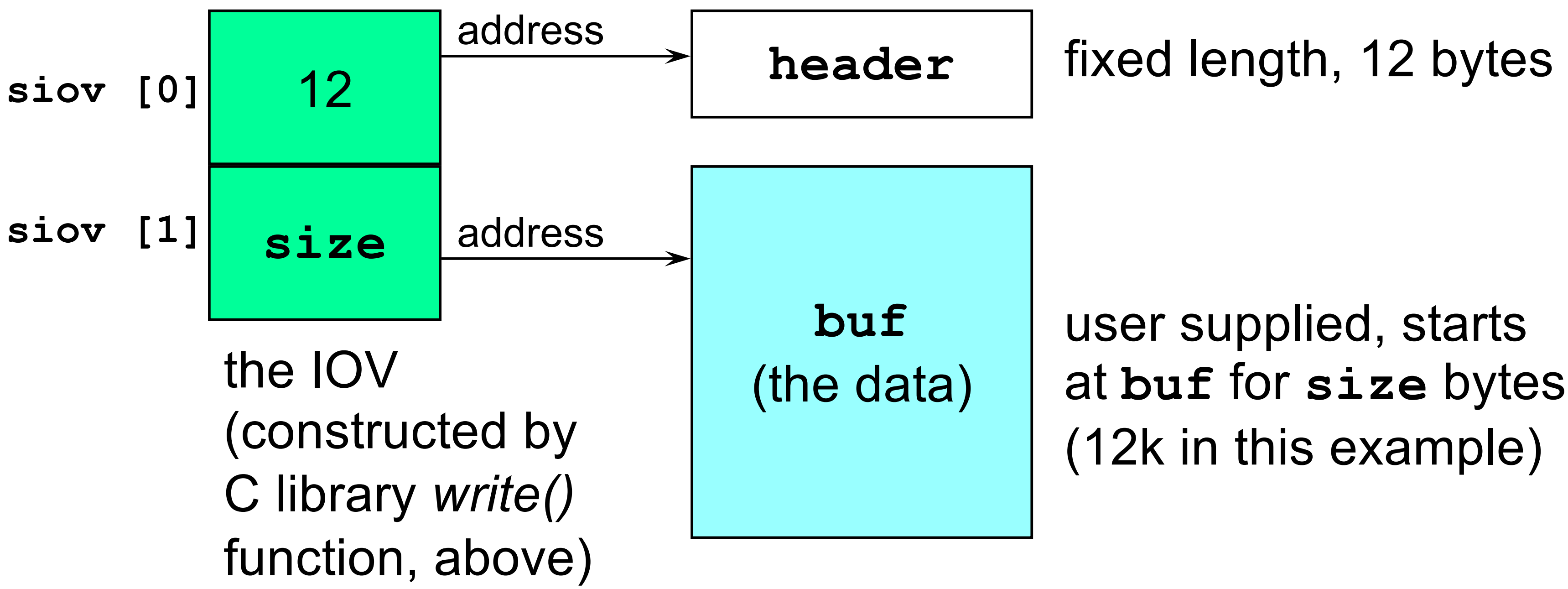
Messaging - Using IOVs

Variable length message example - client side:

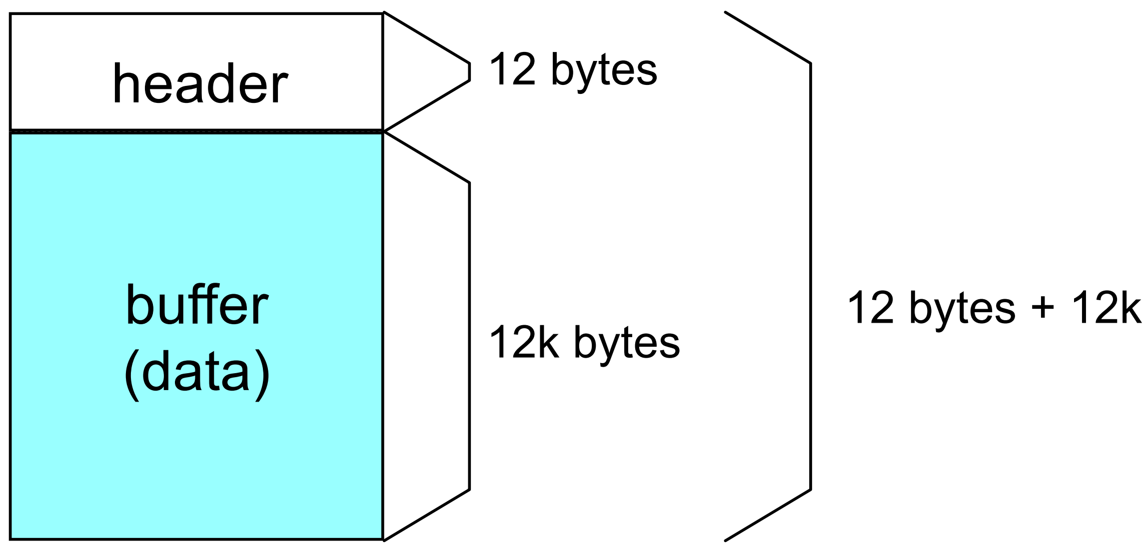
```
write (fd, buf, size);
```

effectively does:

```
header.nbytes = size;
SETIOV (&siov[0], &header, sizeof (header));
SETIOV (&siov[1], buf, size);
MsgSendv (fd, siov, 2, NULL, 0);
```



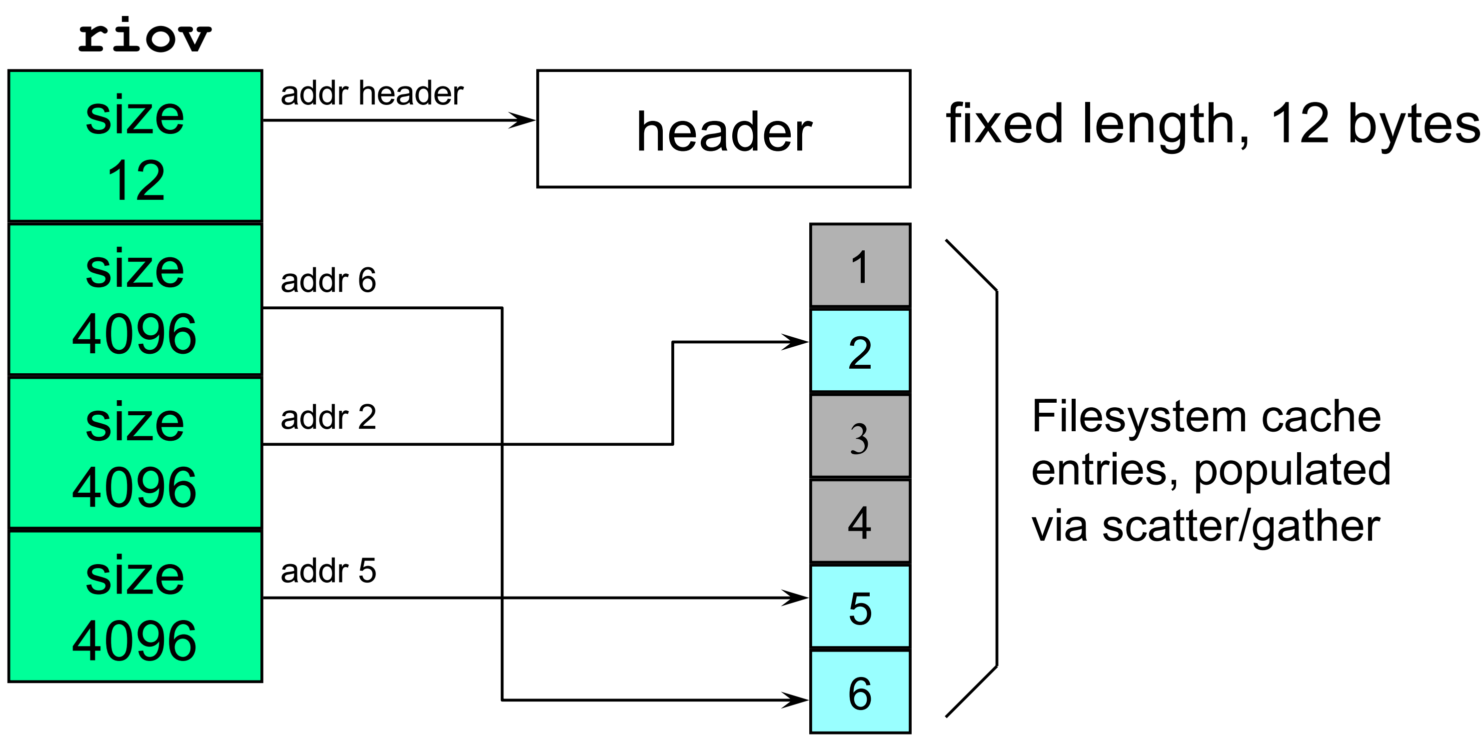
What actually gets sent:



A (logically) contiguous block of bytes

On the server side, what gets received:

```
// assume riov has been setup
MsgReceivev (chid, riov, 4, NULL);
```



In reality, though, we don't know how many bytes to receive, until we've looked at the header:

```
rcvid = MsgReceive (chid, &header,
                    sizeof (header), NULL);
```

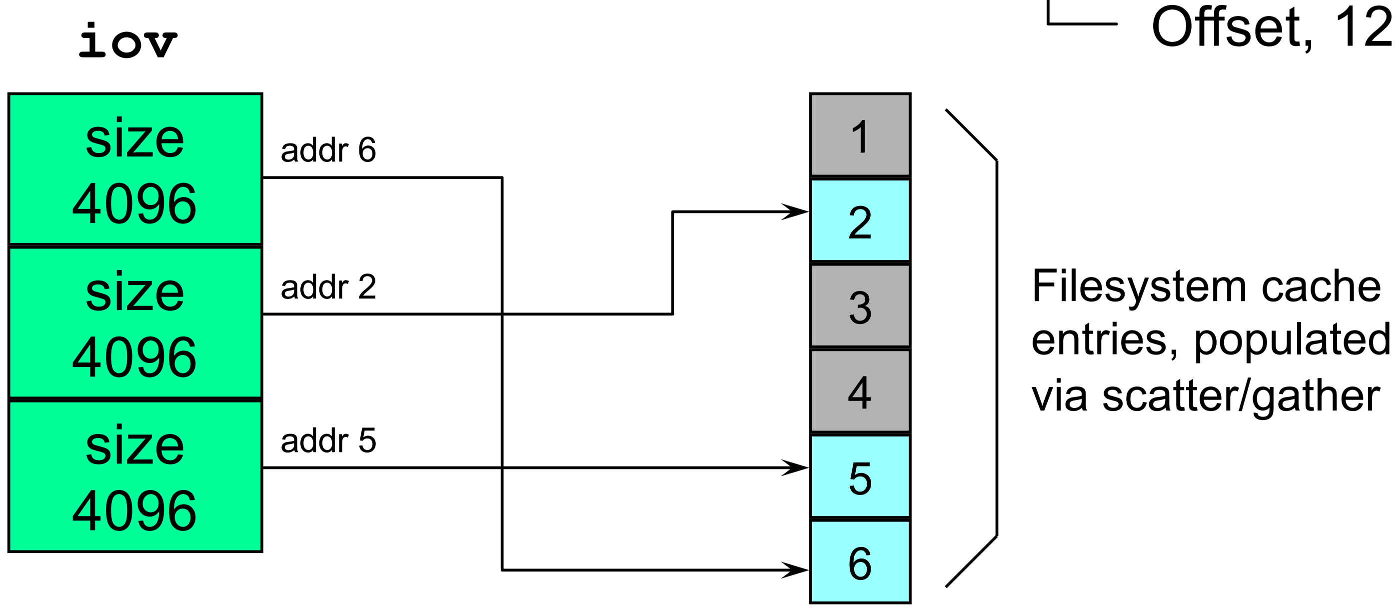
header

In this example, the header would have indicated there was 12k of data to read

Then, we can set up an IOV and read:

```
SETIOV (&iov [0], &cbuf [6], 4096);
SETIOV (&iov [1], &cbuf [2], 4096);
SETIOV (&iov [2], &cbuf [5], 4096);
MsgReadv (rcvid, iov, 3, sizeof(header));
```

Offset, 12 bytes



The *MsgRead()* call:

```
bytes_copied = MsgRead(rcvid, rmsg, rbytes, offset);
```

`rcvid` is the receive ID, provided by the *MsgReceive()* that the server has to do before *MsgRead()*

`rmsg` is a buffer in which the message data is to be received into,

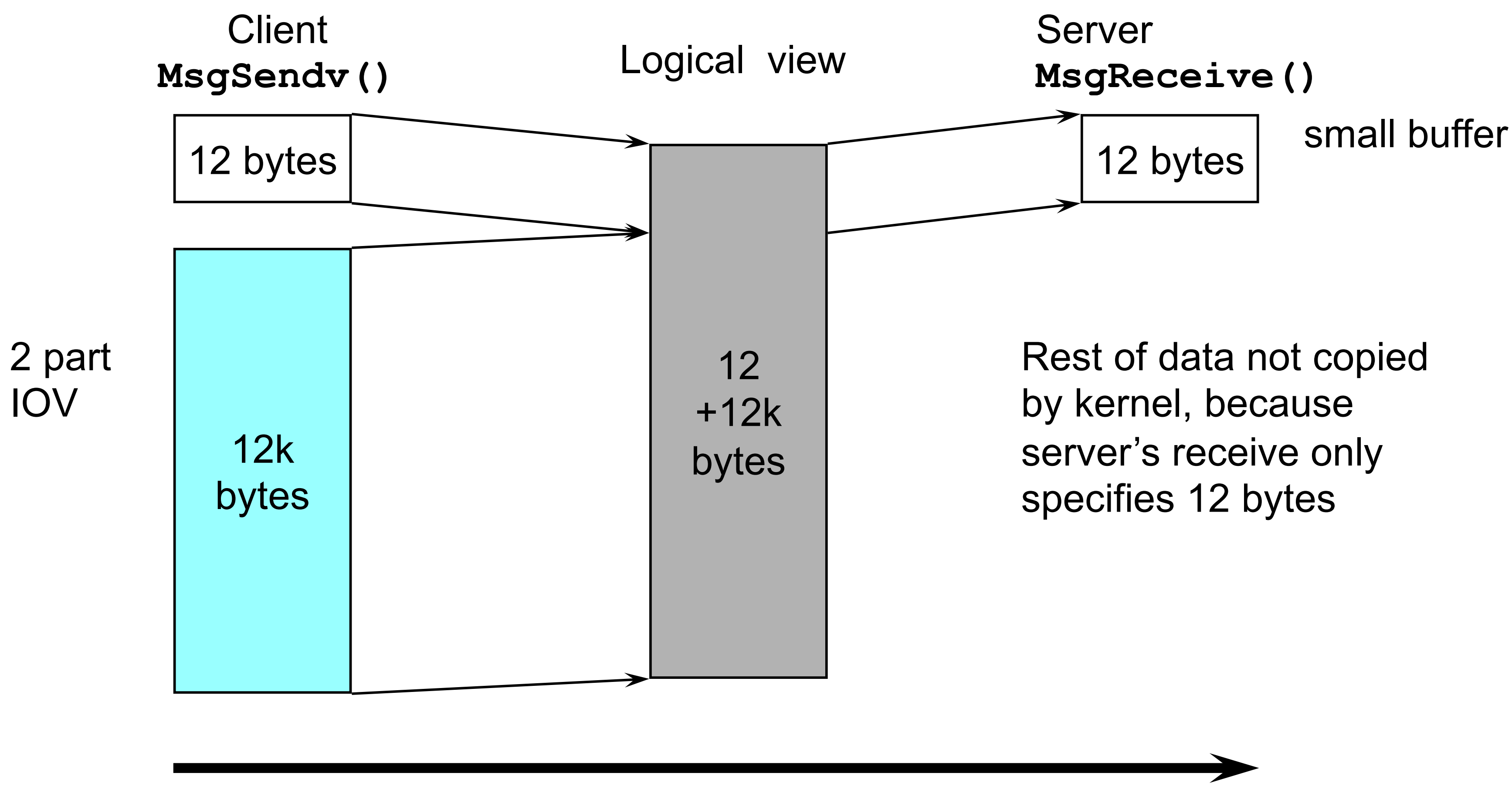
`rbytes` is the number of max. number of bytes to receive in **`rmsg`**,

`offset` is the position within the clients send buffer to start reading from

- allows server to skip header, or data that has already been received/read, or isn't needed

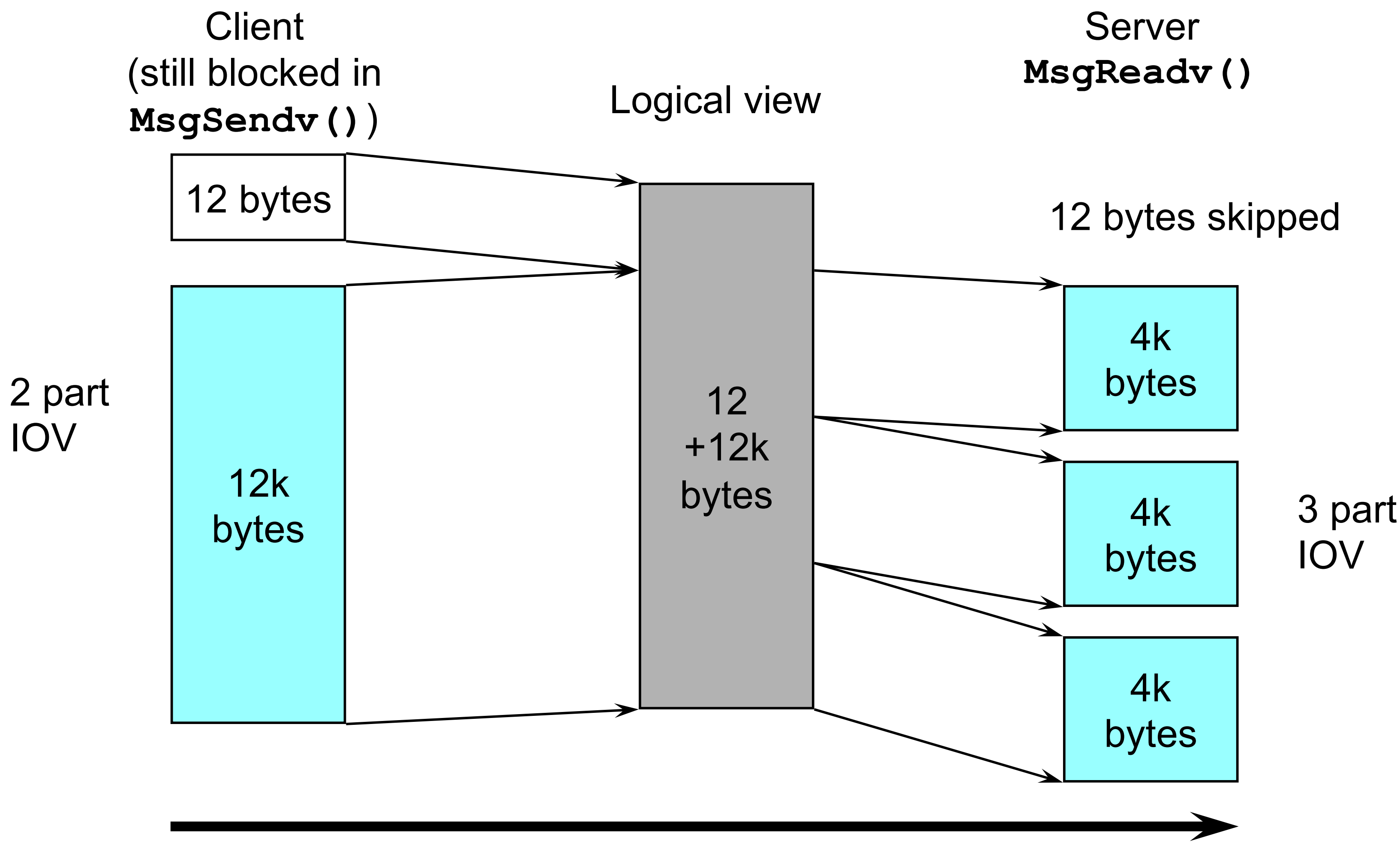
MsgRead() and *MsgReadv()* return the number of bytes copied from client to server

From client to server:



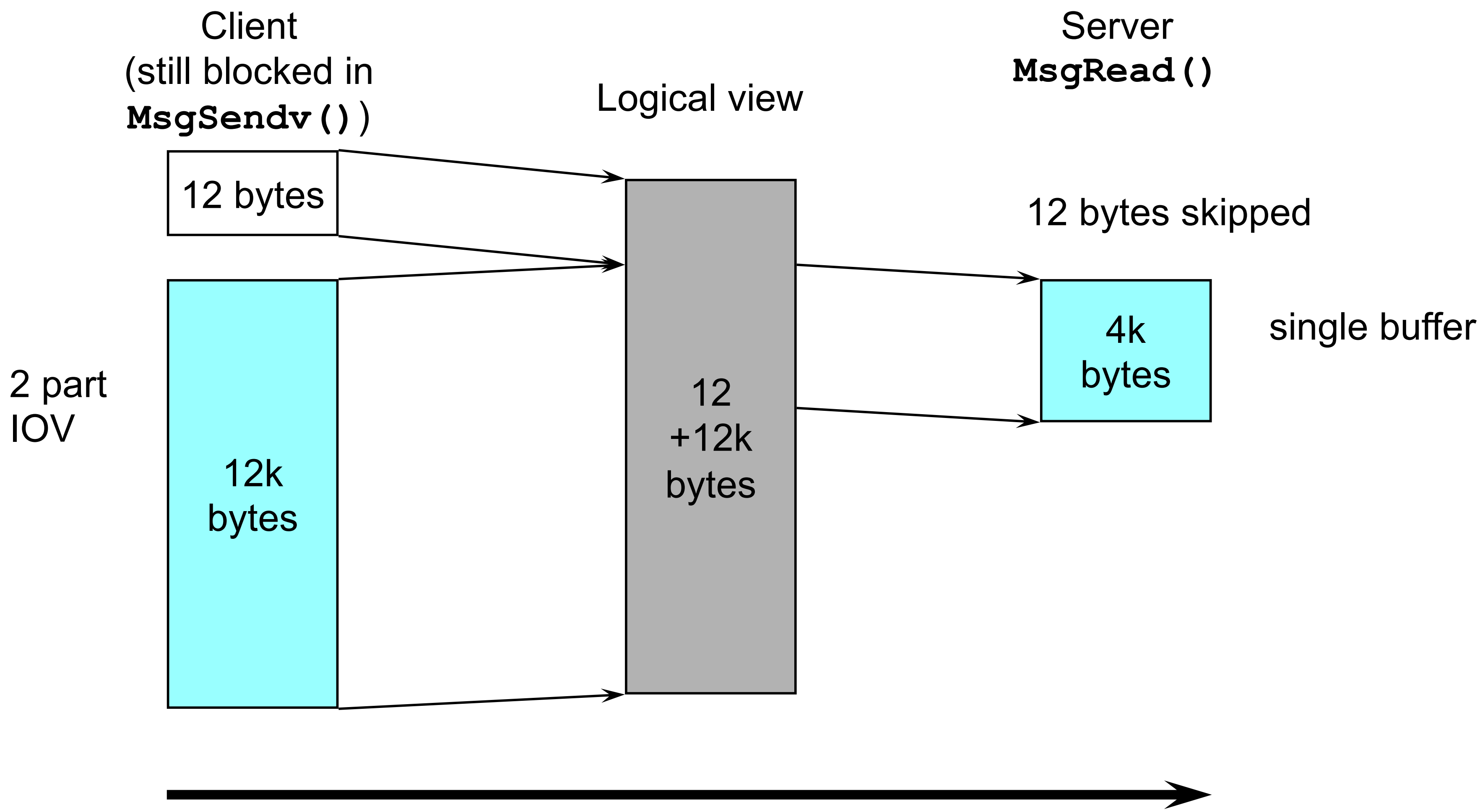
Data copied from client's address space to server's address space by kernel

Continuing from client to server:



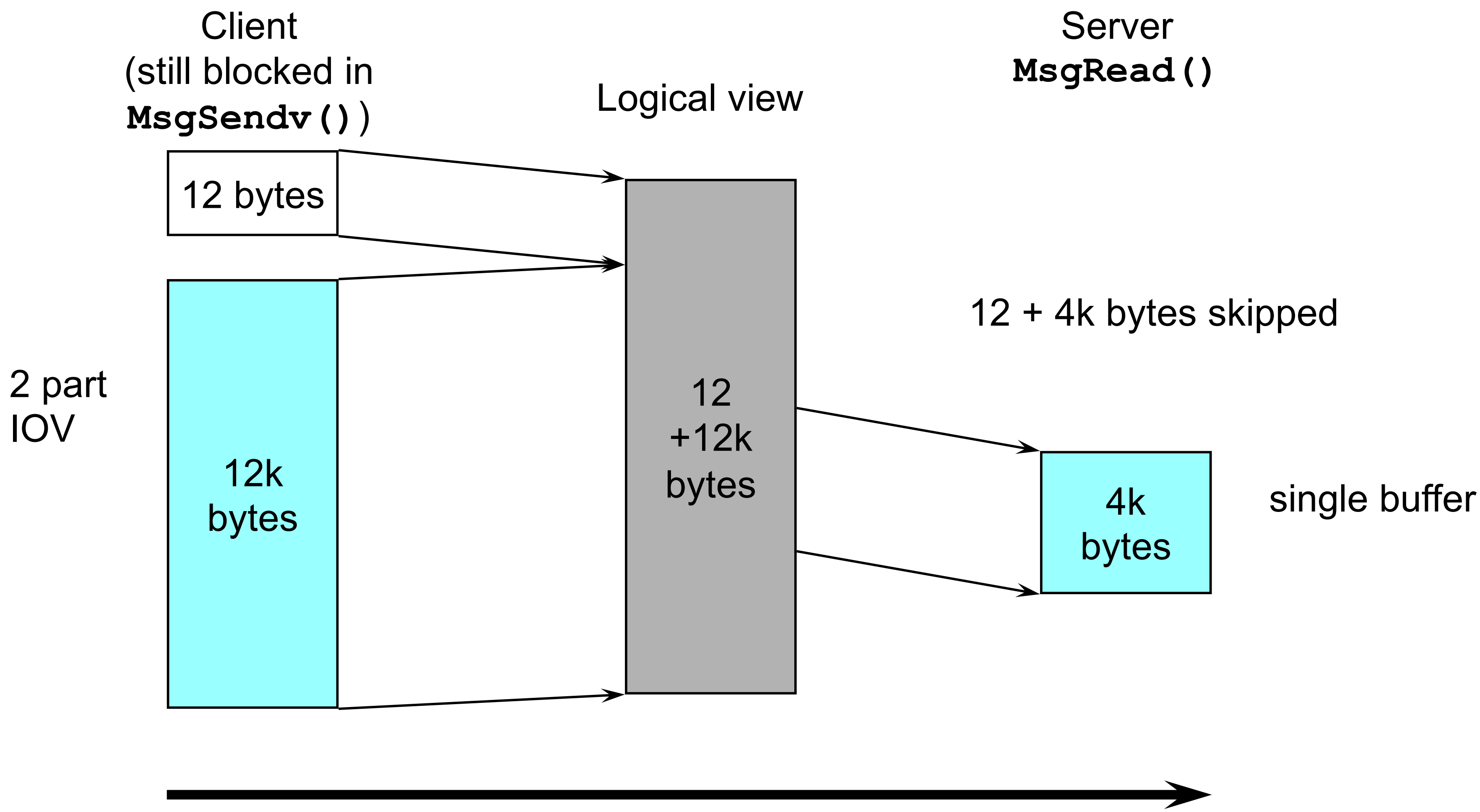
Data copied from client's address space to server's address space by kernel

Or, alternatively:



Data copied from client's address space to server's address space by kernel

And then getting more later:



Data copied from client’s address space to server’s address space by kernel

MsgWrite

For copying from server to client:

```
MsgWrite (rcvid, smsg, sbytes, offset)
MsgWritev (rcvid, siov, sparts, offset)
```

They write bytes from the server to the client, but don't unblock the client.

The data from smsg or siov is copied to the client's reply buffer.

They return the number of bytes actually copied.

To complete the message exchange (i.e., unblock the client), call *MsgReply*()*.

Stressing where the message data goes:

```
MsgSend (coid, txmsg, txbytes, rxmsg, rxbytes);

MsgReceive (chid, rxmsg, rxbytes, &info);

MsgRead (rcvid, rxmsg, rxbytes, offset);

MsgWrite (rcvid, txmsg, txbytes, offset);

MsgReply (rcvid, status, txmsg, txbytes);
```

EXERCISE

IOV messaging exercise:

- extend your **`name_lookup_client.c`** and **`name_lookup_server.c`** files in your **`ipc`** project to use IOV messaging
  - copy them to **`iov_client.c`** and **`iov_server.c`**
  - copy **`msg_def.h`** to **`iov_server.h`**
  - update the **`Makefile`** to build them
- modify the client to send a string from the user, which can vary in length
  - have it create a 2-part IOV message
    - header that says how many bytes are in the string
      - don’t forget about the message type, it is still required
    - a buffer that contains the actual data, it simply follows the header
- modify the server so that receives only the header, and:
  - looks to see how many bytes are in the string
  - allocates enough memory for the string
  - *MsgRead()*s the rest of the string

Topics:

- Message Passing
- Designing a Message Passing System (1)
- Pulses
- How a Client Finds a Server
- Client Information Structure
- Server Cleanup
- Multi-Part Messages
- → Designing a Message Passing System (2)
- Issues Related to Priorities
- Designing a Message Passing System (3)
- Event Delivery
- Shared Memory
- Conclusion

When dealing with large/variable length data carrying messages:

- they should be built as a header followed by the data
  - header will specify amount of data to follow
  - client will generally header/data using an iov, e.g.

```
SETIOV(&iov[0], &hdr, sizeof(hdr) );
SETIOV(&iov[1], data_ptr, bytes_of_data );
MsgSendv(coid, iov, 2, …);
```

- server will generally want a receive buffer large enough to handle all non-data carrying messages
  - can easily do this by declaring the receive buffer to be a union of all message structures
  - use *MsgRead*()* to process large data messages

Topics:

- Message Passing
- Designing a Message Passing System (1)
- Pulses
- How a Client Finds a Server
- Client Information Structure
- Server Cleanup
- Multi-Part Messages
- Designing a Message Passing System (2)
- → Issues Related to Priorities
- Designing a Message Passing System (3)
- Event Delivery
- Shared Memory
- Conclusion

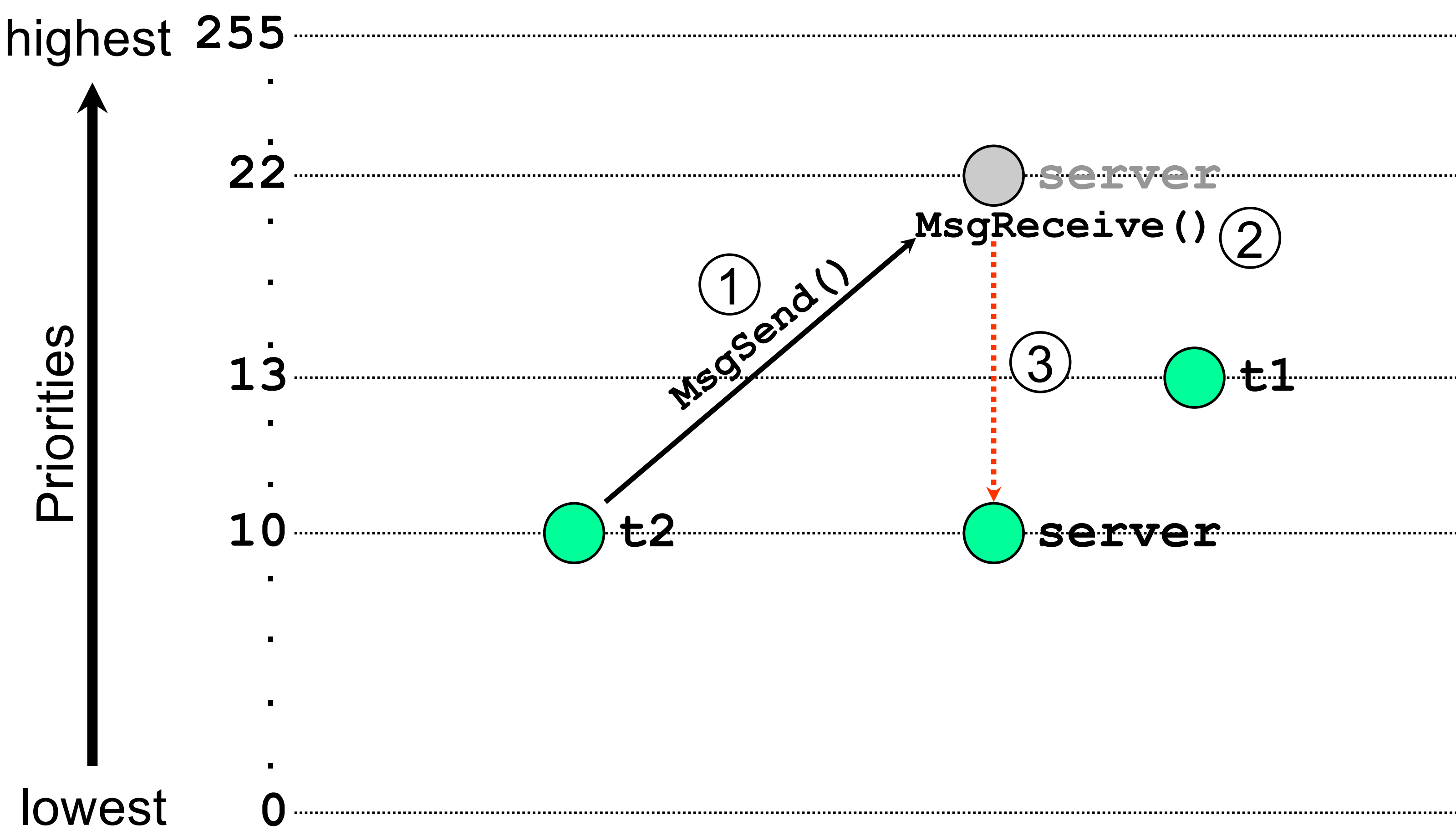
Priority Queueing

If the server calls *MsgReceive()* and:

- there are several clients SEND blocked:
  - the message from the highest priority SEND blocked client is received
    - the server thread runs at this priority while handling this message
  - if multiple clients have the same priority, the one that has been waiting longest is handled
  - this follows the same rules as scheduling
- there are multiple pulses queued:
  - pulses are delivered in priority, then time, order
  - the server thread runs at the pulse priority
- both pulses and messages are waiting:
  - they are delivered in priority, then time, order
  - messages are not favored over pulses, nor the reverse

Priority Inheritance - Decrease

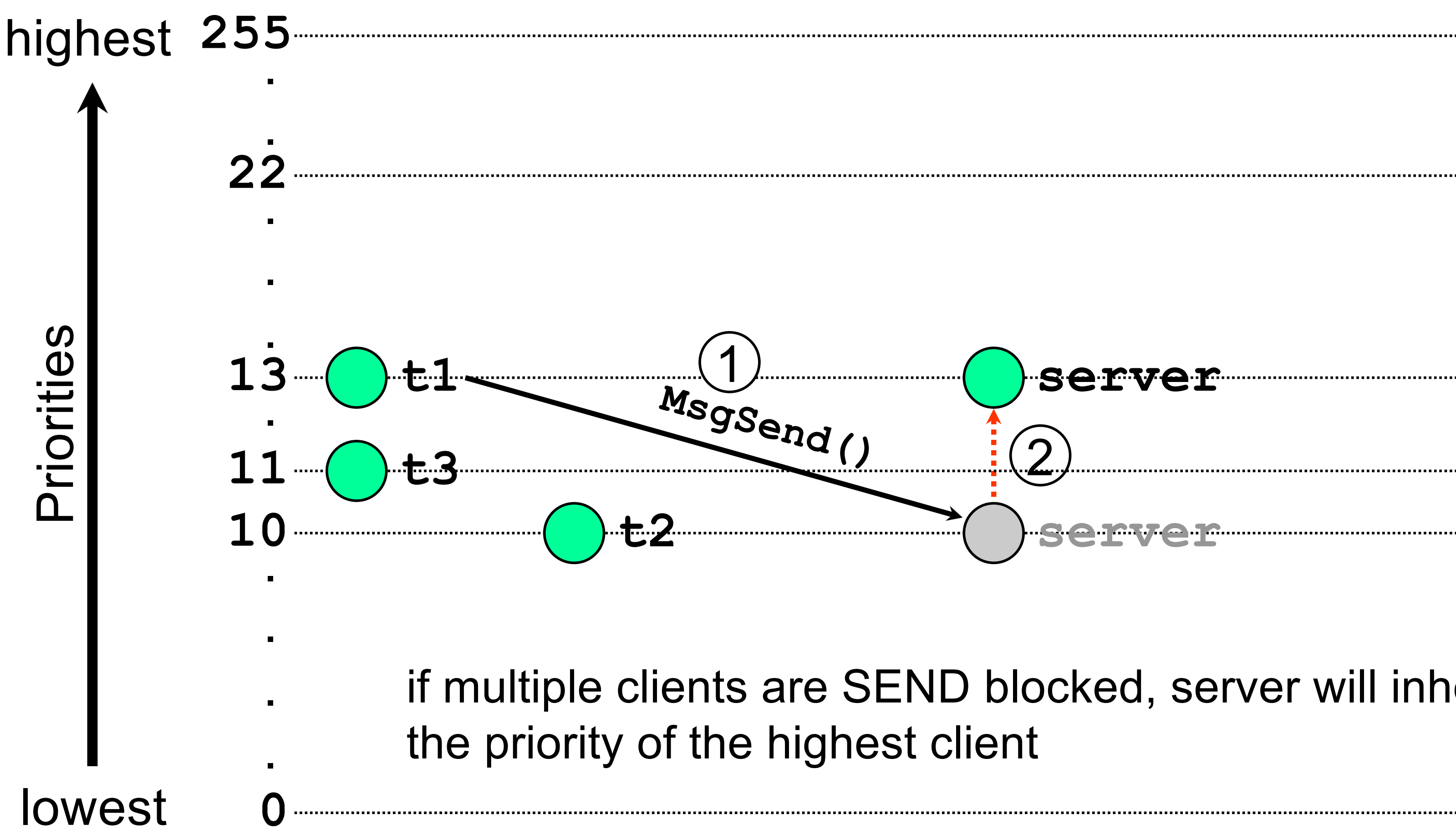
The server drops in priority when it receives a message:



continued...

Priority inheritance - Increase

The server is boosted in priority when the client sends a message:

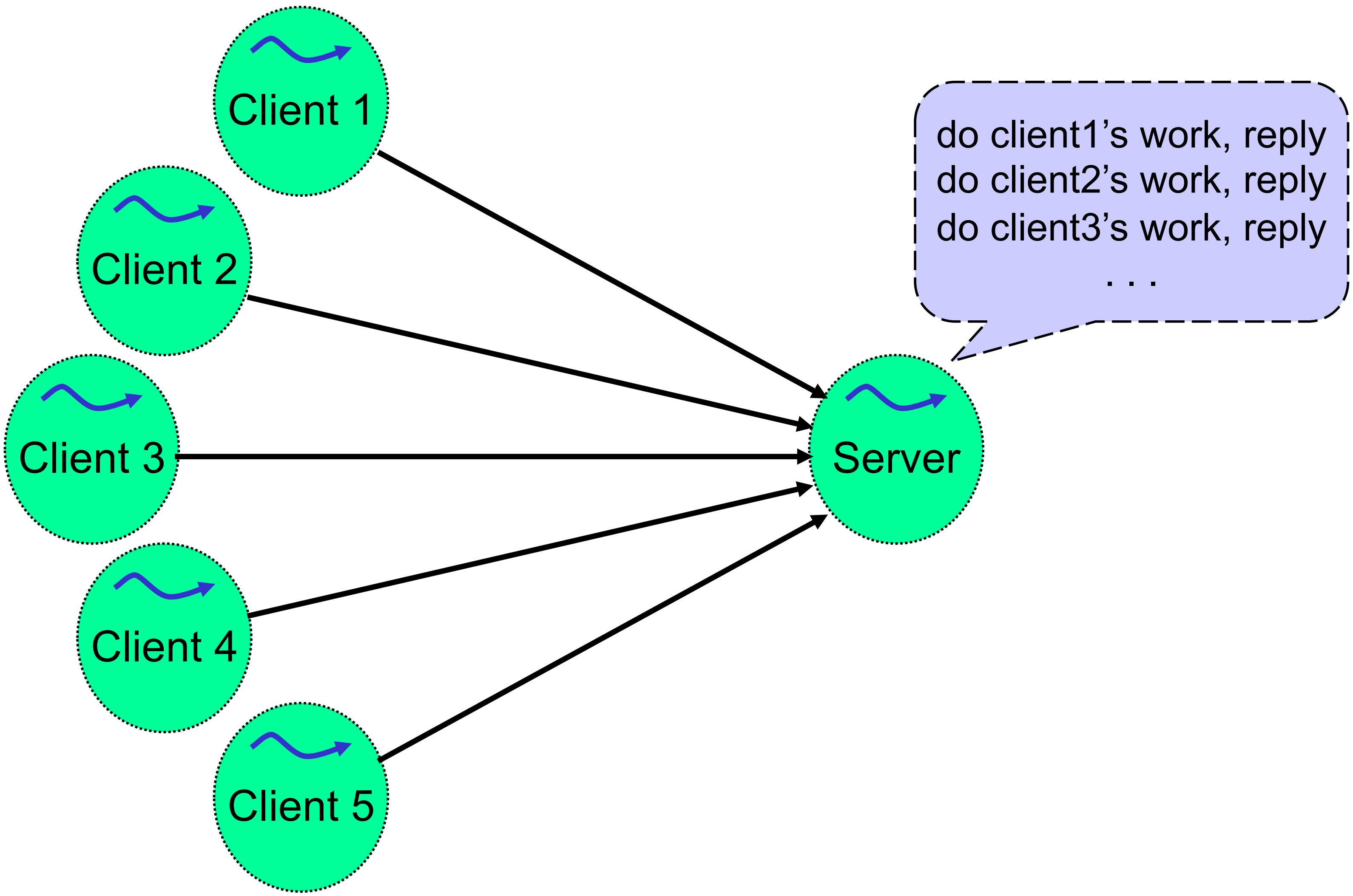


if multiple clients are SEND blocked, server will inherit the priority of the highest client

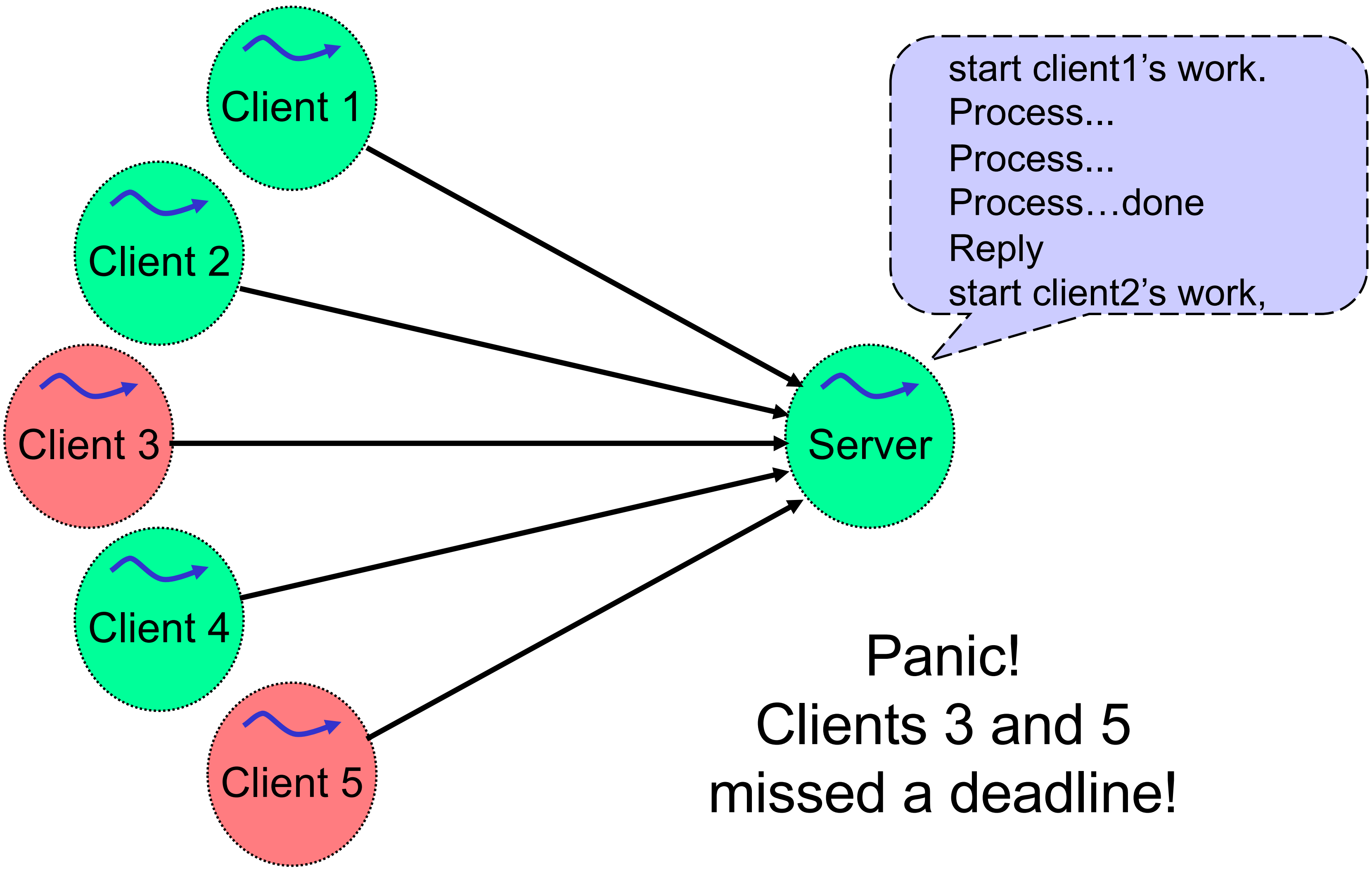
Topics:

- Message Passing
- Designing a Message Passing System (1)
- Pulses
- How a Client Finds a Server
- Client Information Structure
- Server Cleanup
- Multi-Part Messages
- Designing a Message Passing System (2)
- Issues Related to Priorities
- Designing a Message Passing System (3)
  - Server Designs
  - Deadlock Avoidance
- …

Typical server:

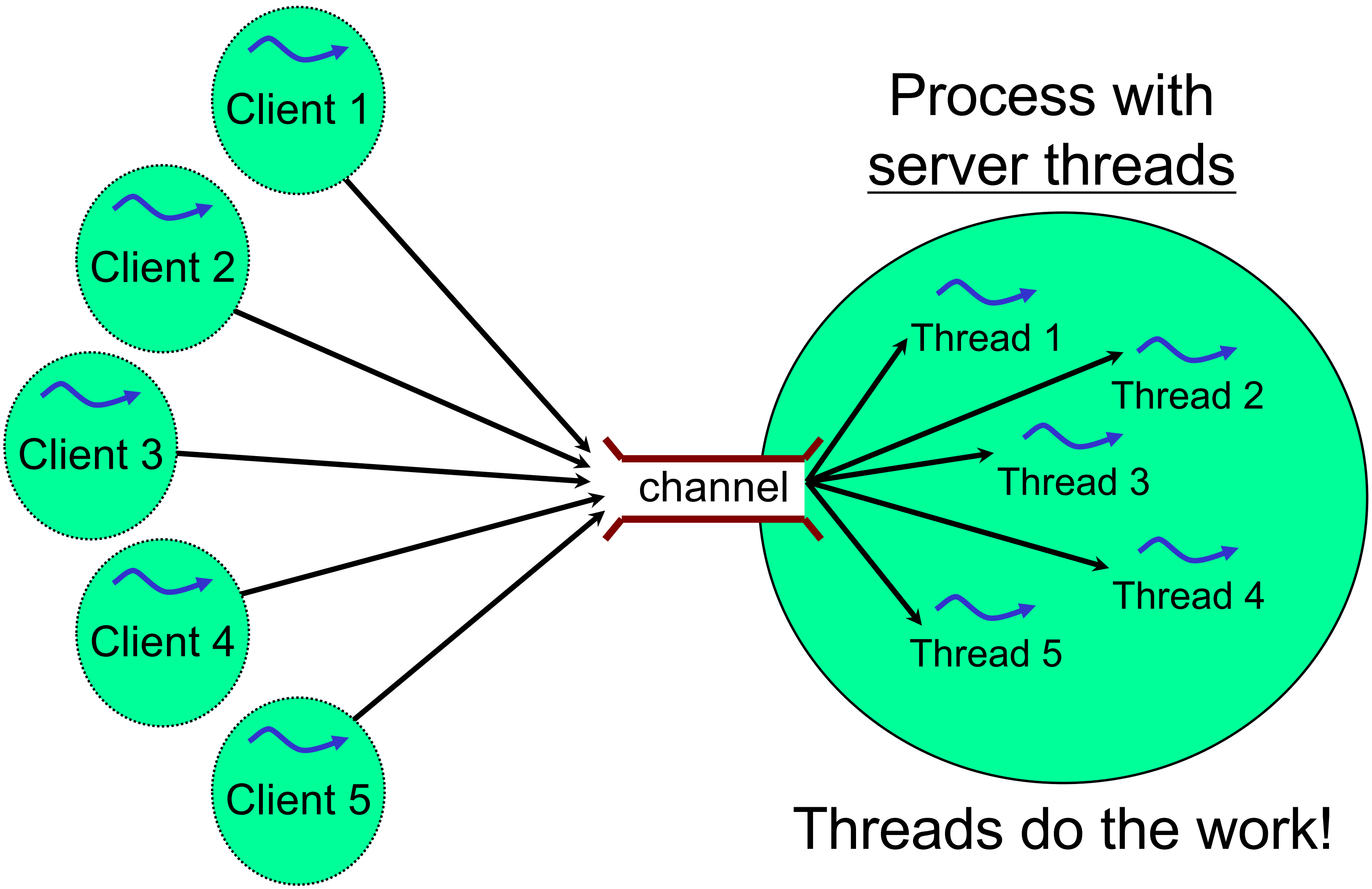


If the request takes a long time:



Panic!
Clients 3 and 5
missed a deadline!

The server can start up some threads:



Threads do the work!

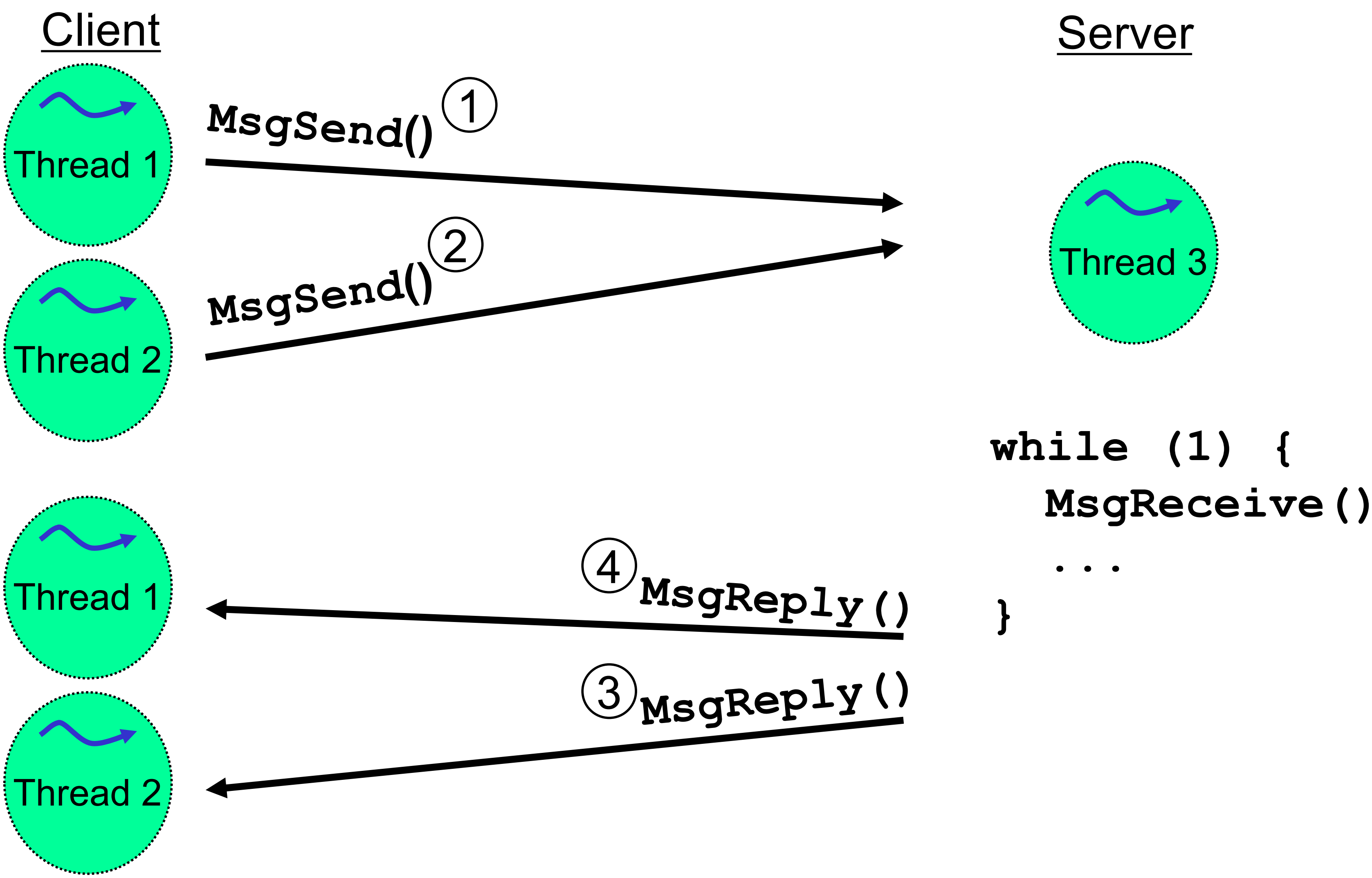
Multithread Model

The multithread model:

- threads all use the same chid to receive messages from clients
- threads inherit the priority of their respective clients
- in the case of a multicore system, the server can truly handle multiple requests at the same time

Messaging - Delayed reply

The server doesn't need to reply immediately:



Delayed reply example:

Client

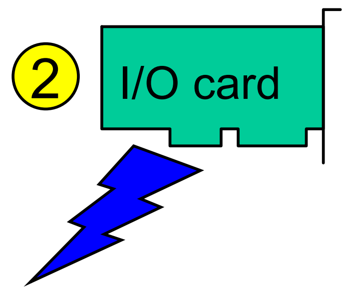
1

```
// request data
MsgSend(...);
```

3 we give the data to the client

Server

```
MsgReceive(...)
{
   // add request to a
   // queue

   // go back to MsgReceive
   // without MsgReply
}

handling_code(...)
{
   ...
}
```



something happens, e.g.
hardware interrupt
-this causes a thread to run some code

Delayed reply example:

```
   pid tid name                prio STATE       Blocked
     1   1 /procnto-smp-instr    0f READY
     1   2 /procnto-smp-instr  255r RECEIVE     1
     1   3 /procnto-smp-instr    1r RECEIVE     3
     1   4 /procnto-smp-instr   10r CONDVAR     (0xffff
     1   5 /procnto-smp-instr   10r CONDVAR     (0xffff
     1   6 /procnto-smp-instr  255r RECEIVE     6
     1   7 /procnto-smp-instr   10r READY
     1   8 /procnto-smp-instr   10r READY
143375   1 sbin/devc-pty        10r RECEIVE     1
151570   1 usr/sbin/sshd        10r SIGWAITINFO
155665   1 proc/boot/pidin      10r REPLY       1
159763   1 usr/bin/more         10r REPLY       32777
172042   1 proc/boot/ksh        10r REPLY       143375
```

shell is blocked, waiting for a reply from pid 143375

pid 143375 (the server) is in the RECEIVE state,
waiting for another message, without having replied to the shell

Topics:

Message Passing

Designing a Message Passing System (1)

Pulses

How a Client Finds a Server

Client Information Structure

Server Cleanup

Multi-Part Messages

Designing a Message Passing System (2)

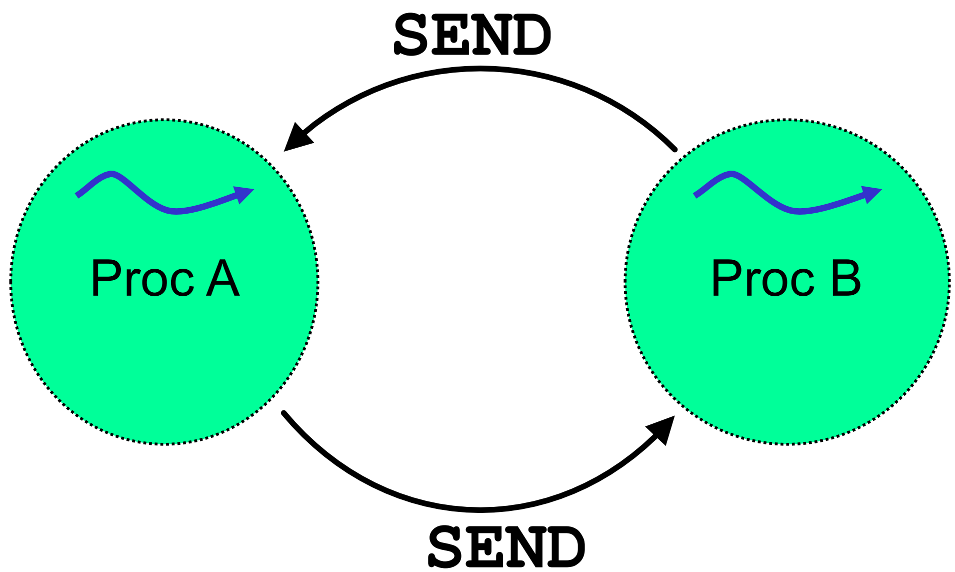
Issues Related to Priorities

Designing a Message Passing System (3)

- **Server Designs**
- **Deadlock Avoidance**

…

What happens if two processes need to send each other data?

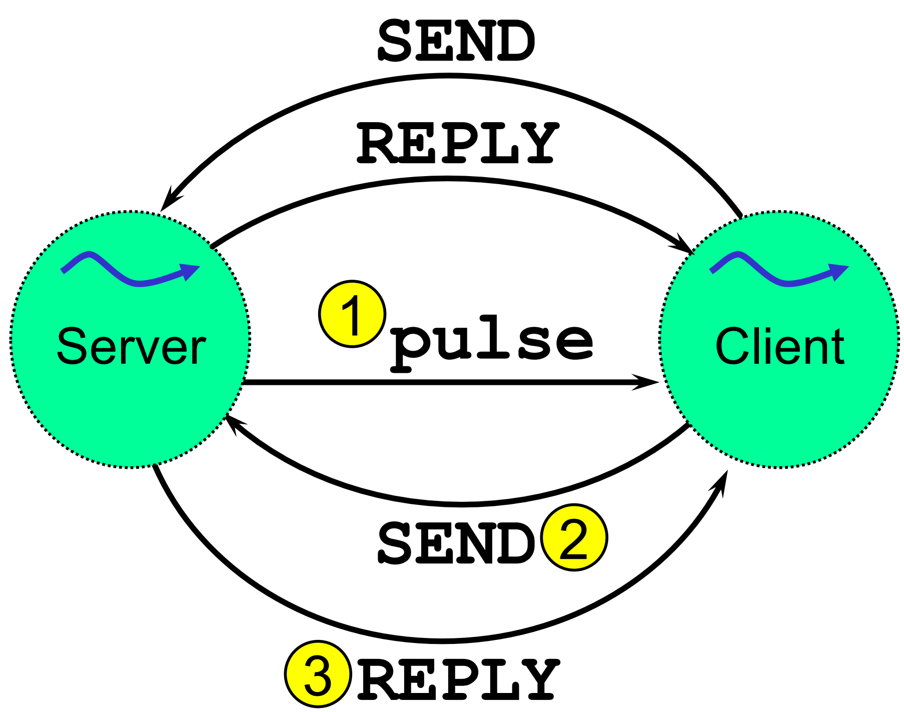


– if two processes SEND to each other, they will be blocked, waiting on each other's reply
  - this is a "DEADLOCK"
  - QNX does not detect this

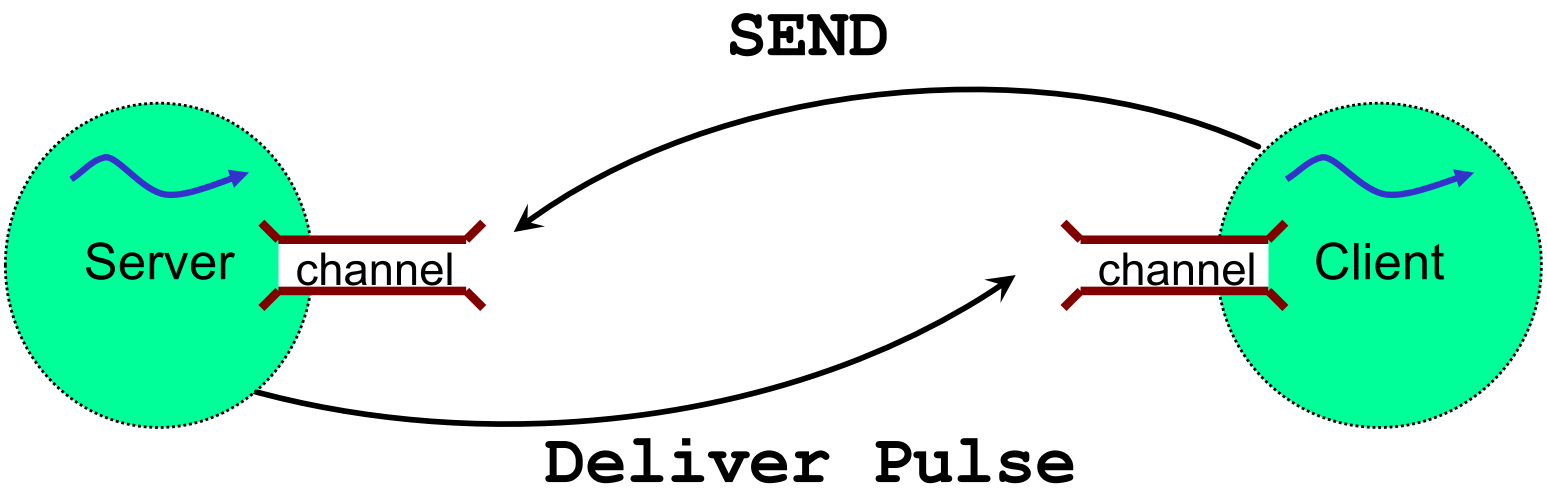
Designing with messages - Deadlock avoidance

We can have the client do all the blocking sending

- the server will use a non-blocking pulse instead
- when the client gets the pulse, it will send the server a message asking for data

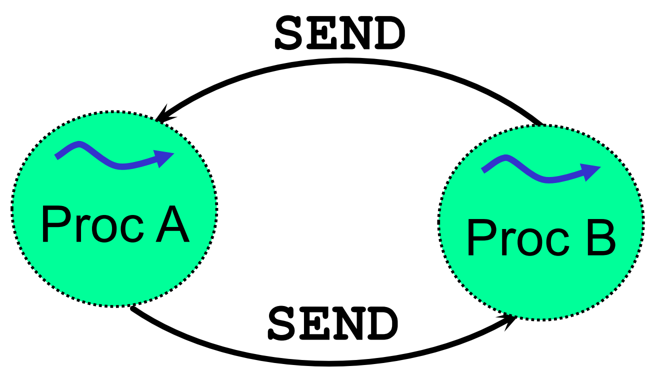


A client can have a channel:

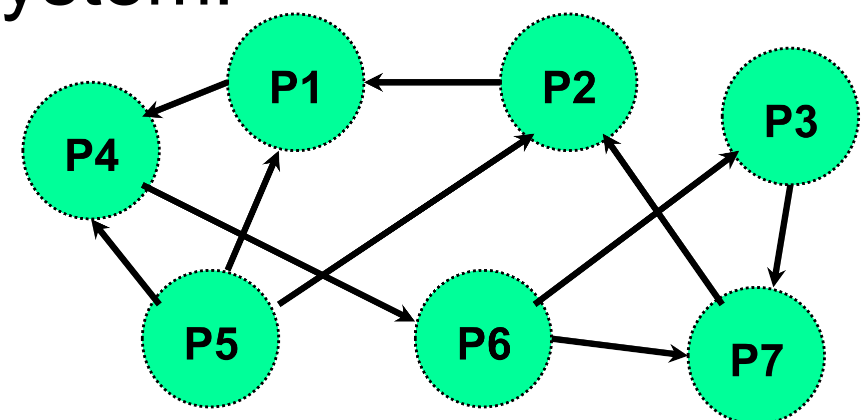


Designing with Messages - Deadlock avoidance

If you only have two processes:

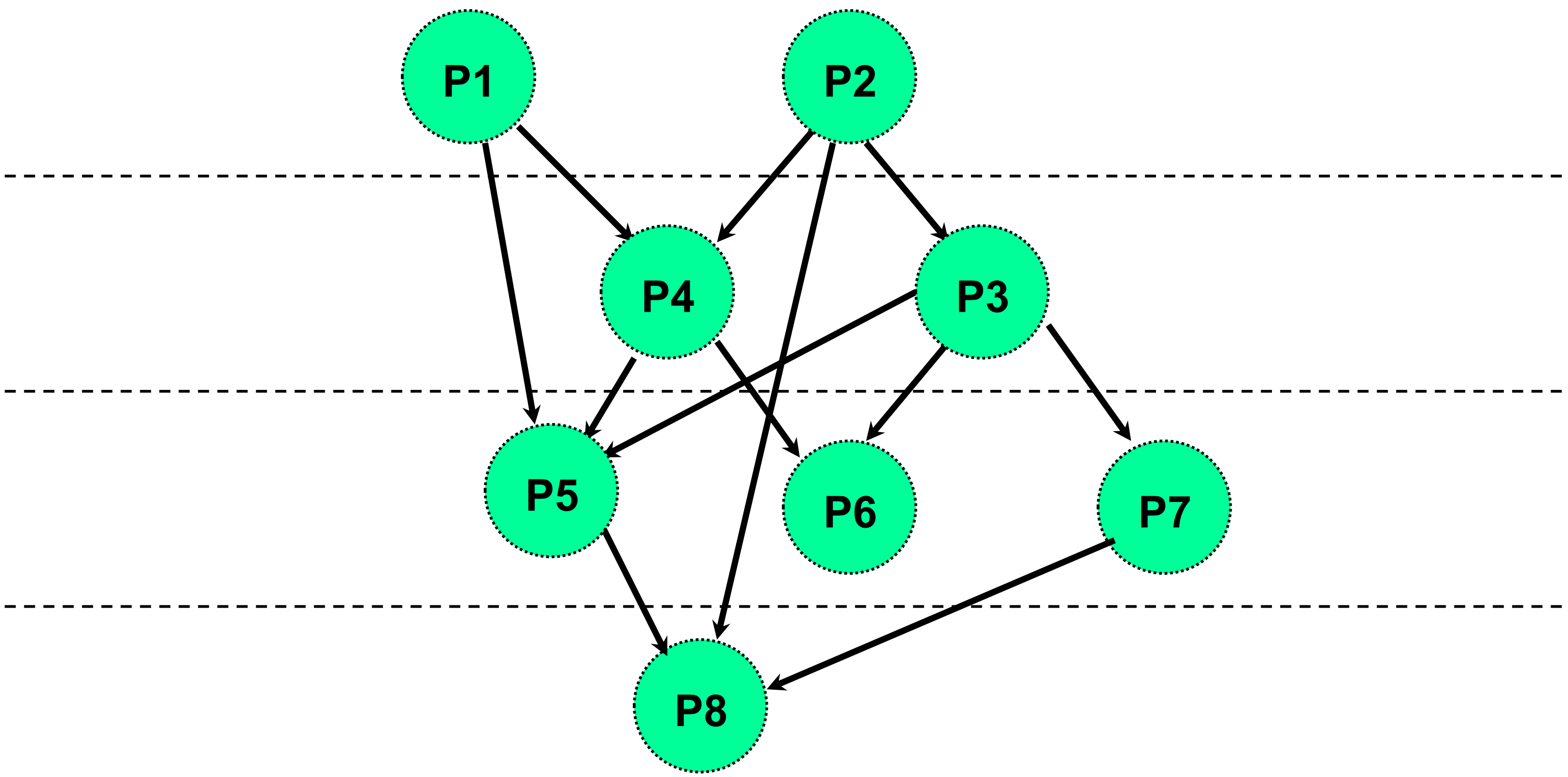


recognizing the potential for deadlock is easy, but in a complex system:



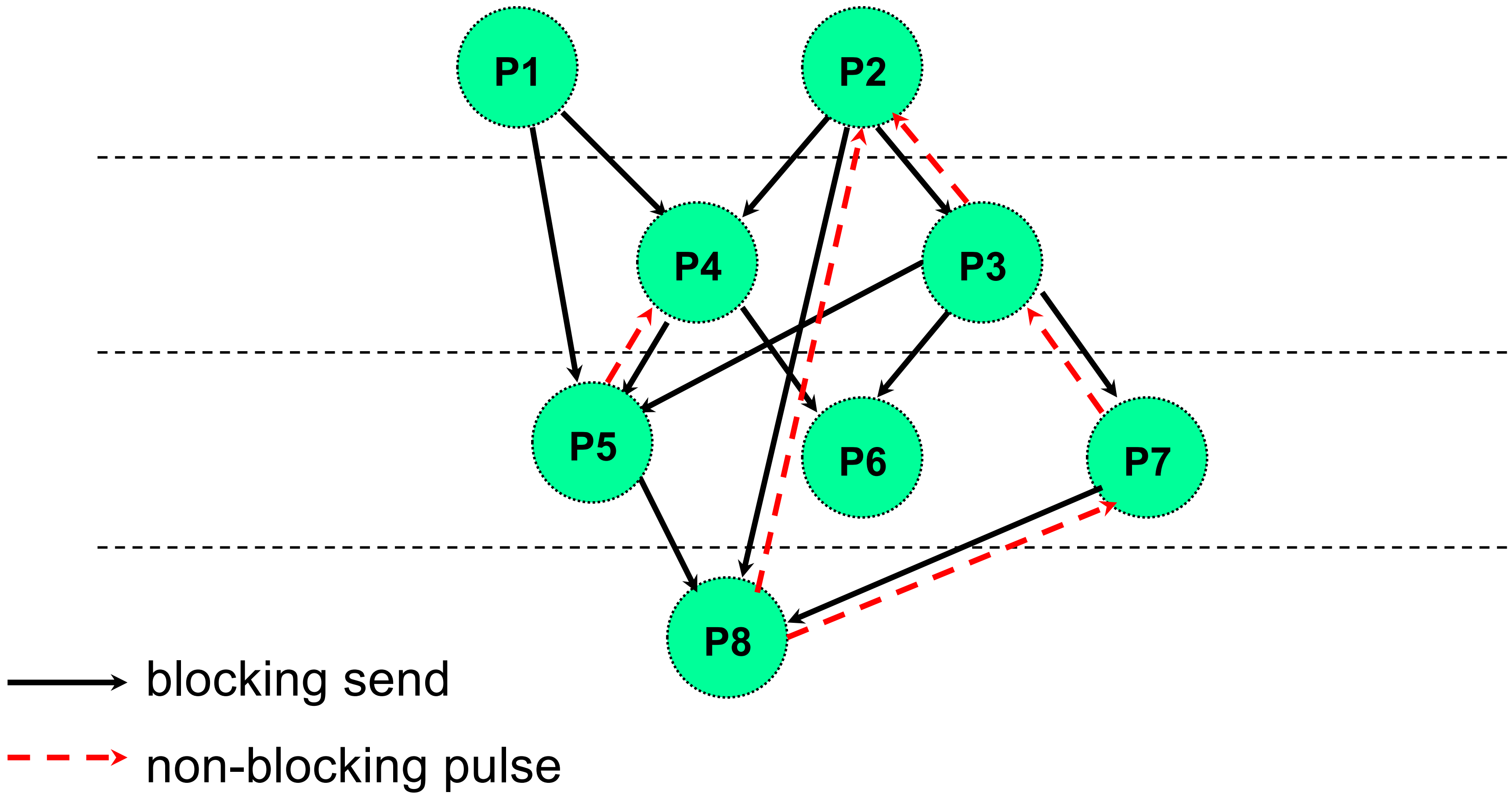
It is much more difficult...

Solution: a Send hierarchy



- sends always go downwards so there can never be a deadlock
- if two processes on the same level need to communicate, just create another level

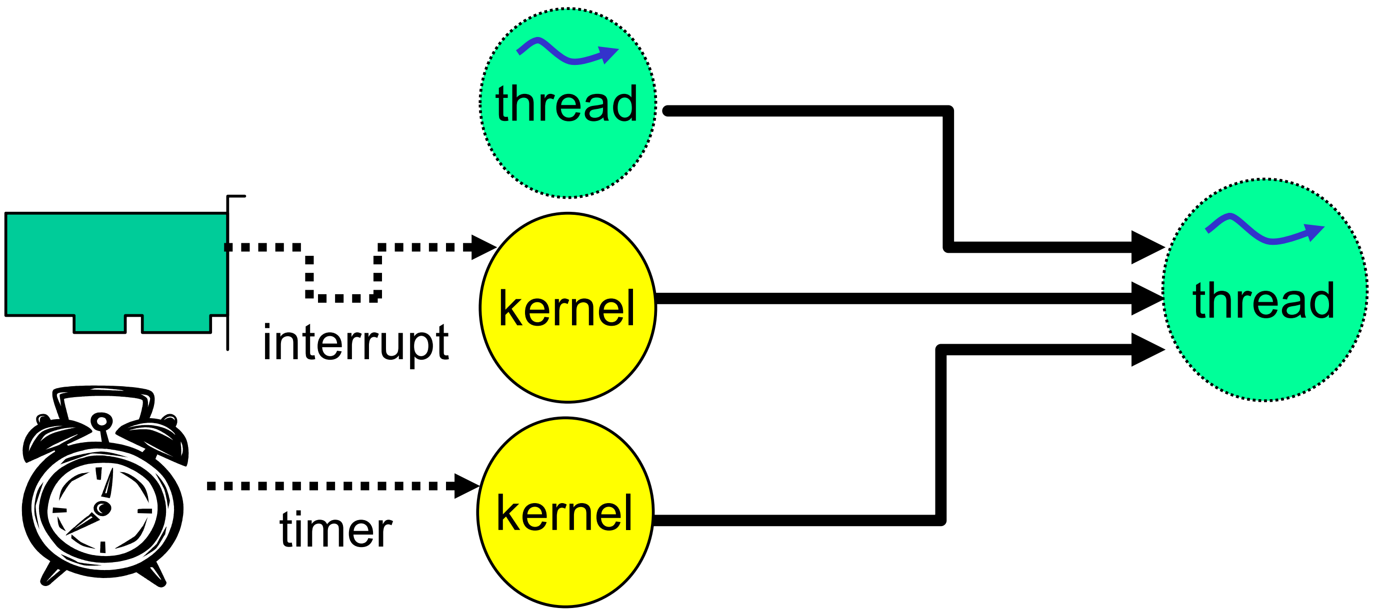
Non-blocking pulses are used for upward-bound notification:



Topics:

Events are a form of notification:

- can be:
  - thread to thread
  - kernel to thread
    - for notification of hardware interrupt
    - for notification of timer expiry



Events:

- can come in various forms:
  - pulses
  - signals
  - can unblock an *InterruptWait()* (only for interrupt events)
  - others
- event properties are stored within a structure: **`struct sigevent`**
- recipient/client usually initializes event structure to choose which form of notification it wants
  - **`struct sigevent`** can be initialized:
    - manually, or
    - using various macros

Macros for initializing an event:

`SIGEV_INTR_INIT(&event);`

- event will unblock an *InterruptWait()* call

`SIGEV_PULSE_INIT(&event, …);`

- event will be a pulse

`SIGEV_SIGNAL_INIT(&event, …);`

- event will be a signal

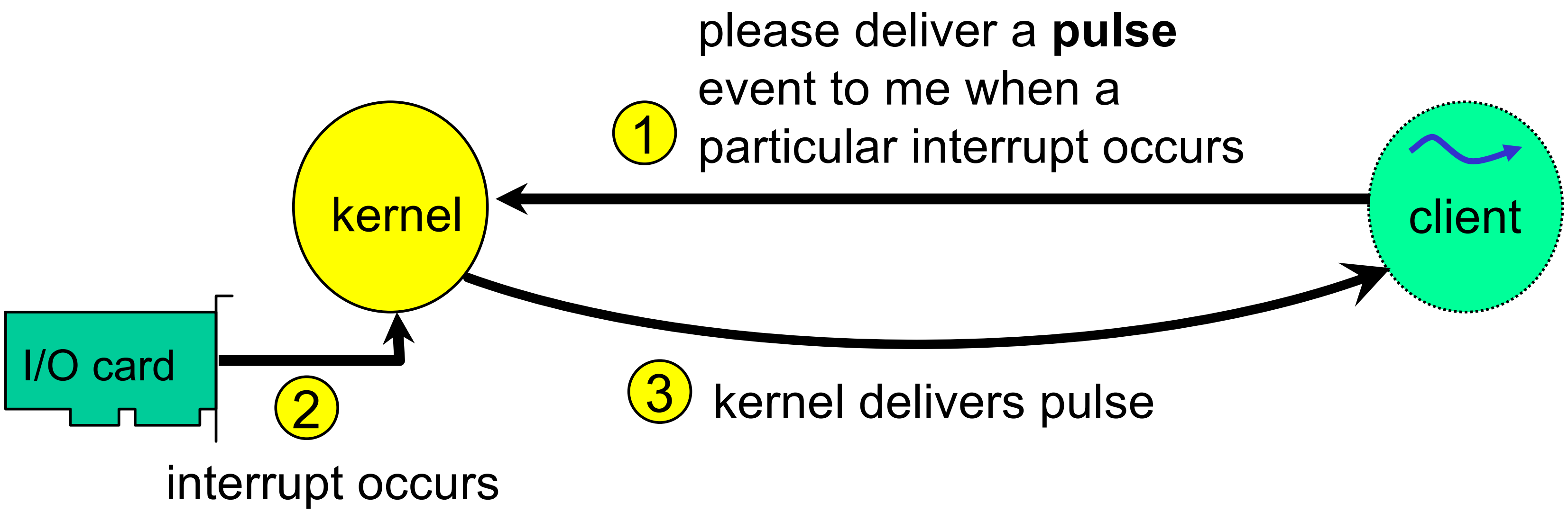
– there are others as well, which are documented in:

- Library Reference→s→sigevent

Example of using a macro to initialize a pulse event:

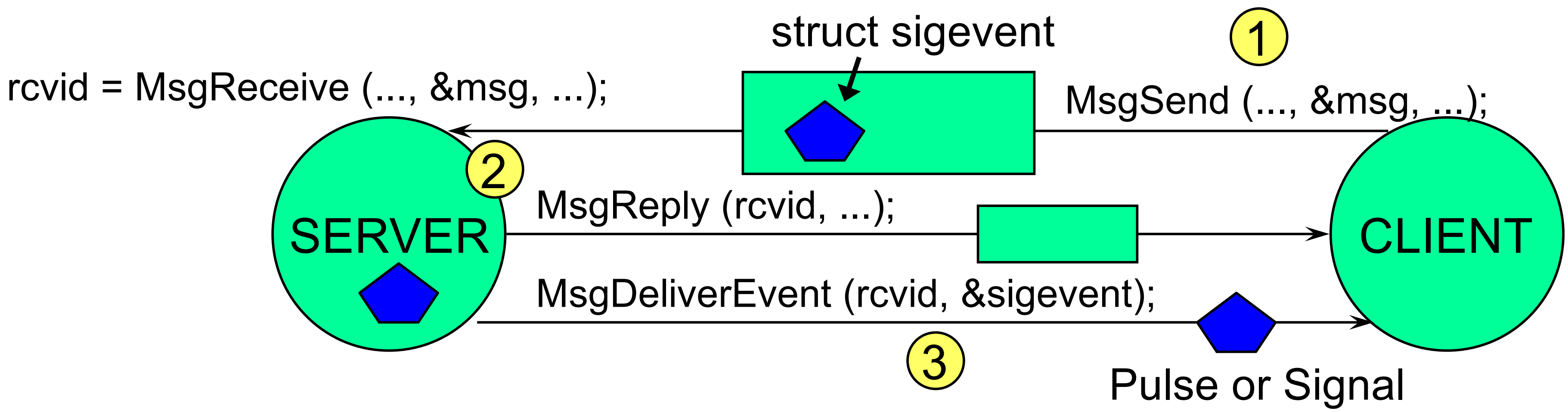
```
chid = ChannelCreate (...);


//connection to our channel
self_coid = ConnectAttach (0, 0, chid, _NTO_SIDE_CHANNEL, flags);

SIGEV_PULSE_INIT(&sigevent, self_coid, MyPriority, OUR_CODE,
        value);
```

Interrupt and Timer Events:



- timer and interrupt events work in a similar fashion
- timer and interrupt handling are covered further in their respective course sections

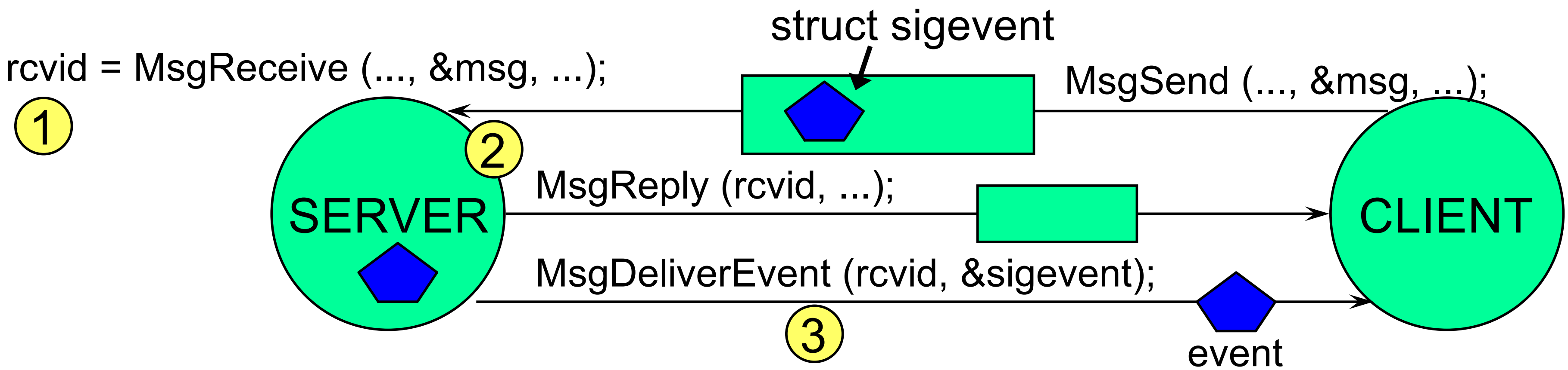
Thread to thread events:



1. client initializes event structure and sends it along with some request to the server
2. server receives, stores the event description somewhere, and responds with "I'll do the work later"
3. when the server completes the work, it delivers the event to the client. the client receives the event and then can send another message asking for the results of the work

☞ server never needs to know what form the event will take, *MsgDeliverEvent()* takes care of it

MsgDeliverEvent() uses the rcvid:



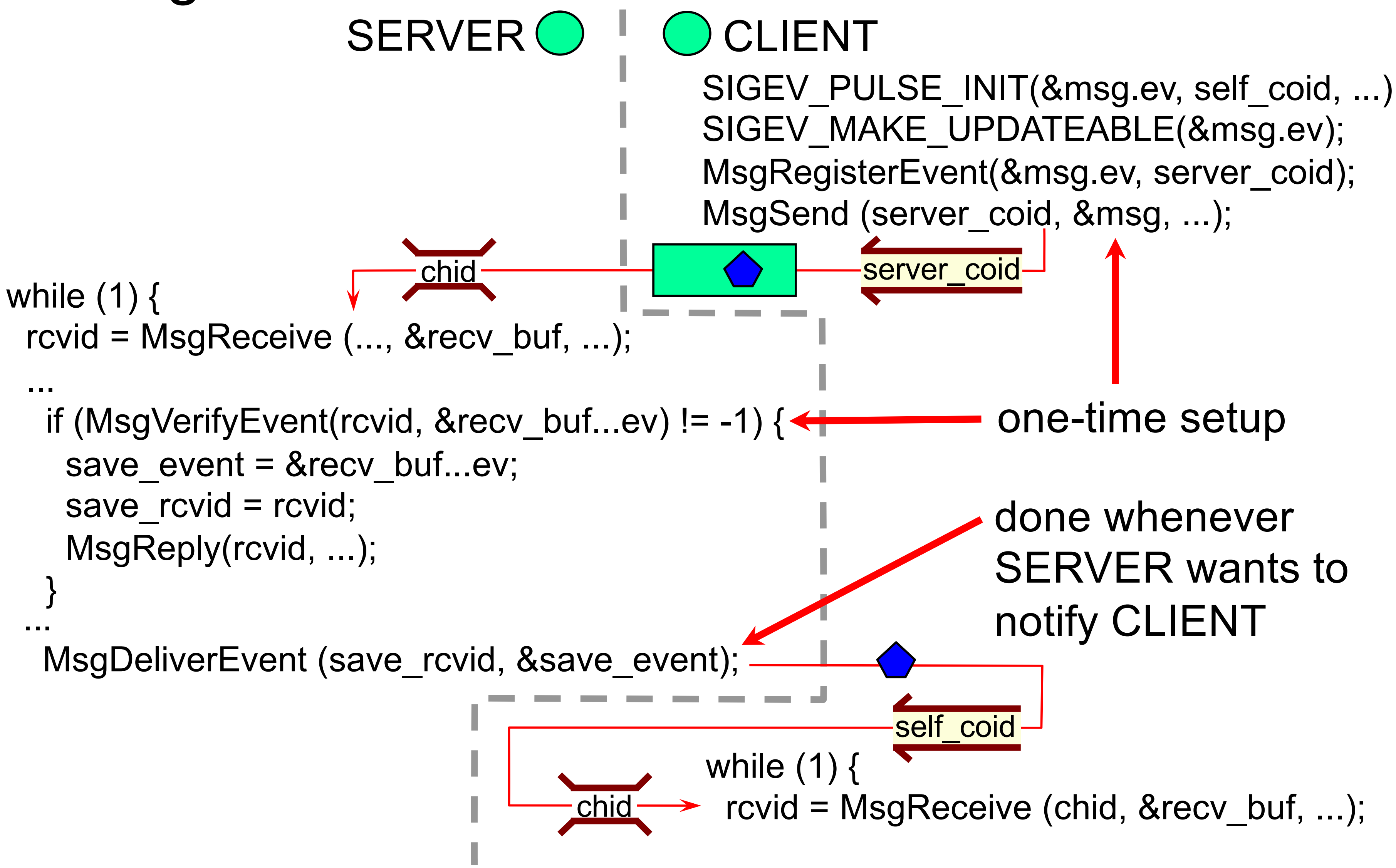
1. server got the rcvid as the return value from the *MsgReceive()*
2. server uses it to *MsgReply()* to the client
3. and then later, the server uses the **`rcvid`** for *MsgDeliverEvent()*

Is the rcvid usable after the *MsgReply()*?

Yes, it contains enough information for the kernel to use to find the client, so we can store it away for use when needed

Event Delivery Example

Adding a few more details:



Event Delivery – Updating events

Generally servers don't modify client's events:

- there may be cases where it would be helpful for the server to pass back information in the event
  - usually done in the **`sigev_value`** field
- client must flag that this is allowed
  - this is done by setting **`SIGEV_FLAG_UPDATEABLE`**
  - tells the server that the client is ok with value being changed
  - tells the kernel that the server is allowed to change the value
    - if not set, client will get the value from when the event was registered
  - usually done with convenience macro:

```
SIGEV_MAKE_UPDATEABLE(&event)
```

- server checks updateable flag in event:

```
if (event.sigev_notify & SIGEV_FLAG_UPDATEABLE)
```

Registering Events

For inter-process delivery you must also register your events:

`MsgRegisterEvent(&event, server_coid)`

- turns your event into a registered event
- only the server that **`server_coid`** connects to may deliver this event
  - a **`server_coid`** of -1 allows any server to deliver the event (less secure)
  - **`SYSMGR_COID`** is used for events from Proc
- not needed for events from the kernel
  - e.g. from a timer or an interrupt
- when the event is no longer needed, deregister it:
  - **`MsgUnregisterEvent(&event)`**

Event verification: protecting the client from themselves:

- if an error occurs at event delivery time it is too late to tell the client
  - only way server has to communicate is event delivery, but that failed
- can check the event when it is first given to them:
  - *MsgVerifyEvent(rcvid, &event)*
  - if it fails, return an error to the client at notification request time

Events and server clean-up:

- the server must store the rcvid, and possibly other information, on a per-client basis
- this needs to be cleaned up (freed) when client disconnects
- we saw this situation earlier, in the server cleanup section

Exercise:

- See **`event_server.c`** and **`event_client.c`** in the **`ipc`** project
- When finished:
  - **`event_client`** will fill in an event (with a pulse) and give it to **`event_server`**
  - **`event_server`** will save away the rcvid and the event
  - **`event_server`** will deliver the event every 1 second - so **`event_client`** will receive the pulse every 1 second

1 Add code to **`event_client.c`**
  - to format the event
  - register the event

2 Add code to **`event_server.c`** to
  - verify the event
  - save away the event
  - deliver it when appropriate.

continued...

Exercise (continued):

- To make it easier, searching for the word “TODO” comments will show you where to make the changes.

- To test, do the following:

  `event_server`

  `event_client`

- every second, **`event_server`** should print out that it sent the pulse and **`event_client`** should print out that it received the pulse.
- kill and restart **`event_client`**

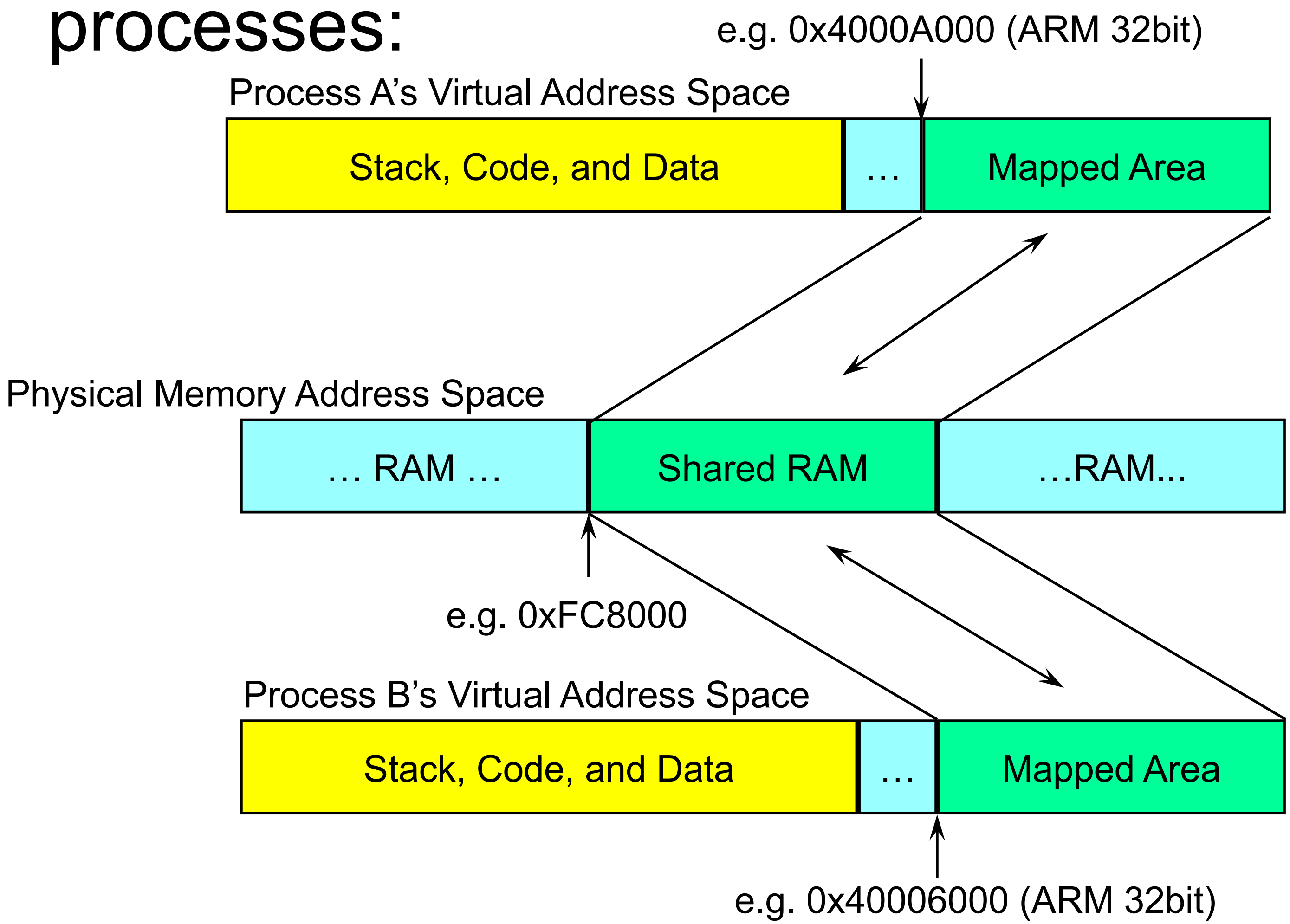
Advanced:

- handle multiple clients in a reasonable fashion:
  - reject a new client if busy, or
  - maintain a client list

Topics:

- Message Passing
- Designing a Message Passing System (1)
- Pulses
- How a Client Finds a Server
- Client Information Structure
- Server Cleanup
- Multi-Part Messages
- Designing a Message Passing System (2)
- Issues Related to Priorities
- Designing a Message Passing System (3)
- Event Delivery
- → Shared Memory
- Conclusion

After setting up a shared memory region, the same physical memory is accessible to multiple processes:



To set up shared memory:

```
fd = shm_open( "/myname", O_RDWR|O_CREAT, 0600 );
```

- name should start with leading / and contain only one /
- using **O_EXCL** can help do synchronization for the case where you have multiple possible creators

```
ftruncate( fd, SHARED_SIZE );
```

- this allocates **SHARED_SIZE** bytes of RAM associated with the shared memory object
  - this will be rounded up to a multiple of the page size, 4K

```
ptr = mmap( NULL, SHARED_SIZE, PROT_READ|PROT_WRITE,
          MAP_SHARED, fd, 0 );
```

- this returns a virtual pointer to the shared memory object
- the next step would be to initialize the internal data structures of the object

```
close(fd);
```

- you no longer need the fd, so you can close it

To access a shared memory object:

```
fd = shm_open( "/myname",O_RDWR, 0);
```

– same name that was used for the creation

```
ptr = mmap( NULL, SHARED_SIZE, PROT_READ|PROT_WRITE,
          MAP_SHARED, fd, 0 );
```

– for read-only access (view), don't use **PROT_WRITE**

– you can gain access to sub-sections of the shared memory by specifying an offset instead of **0**, and a different size

- mapping will be on page-size boundaries, even if offset and size aren't

```
close(fd);
```

– you no longer need the fd, so you can close it

Shared Memory - Cleanup

The allocated memory will be freed when there are no further references to it:

- each fd, mapping, and the name is a reference
- can explicitly close, and unmap:

```
close(fd);
munmap( ptr, SHARED_SIZE );
```

- on process death, all fds are automatically closed and all mapping unmapped
- the name must be explicitly removed:

```
shm_unlink( "/myname" );
```

- during development and testing this can be done from the command line:

```
rm /dev/shmem/myname
```

Problems with shared memory:

- Access issues:
  - pathname collisions
  - security of shared memory objects
- Synchronization issues:
  - readers don’t know when data is stable
  - writers don’t know when it is safe to write

Let’s look at a few solutions...

There are a variety of synchronization solutions:

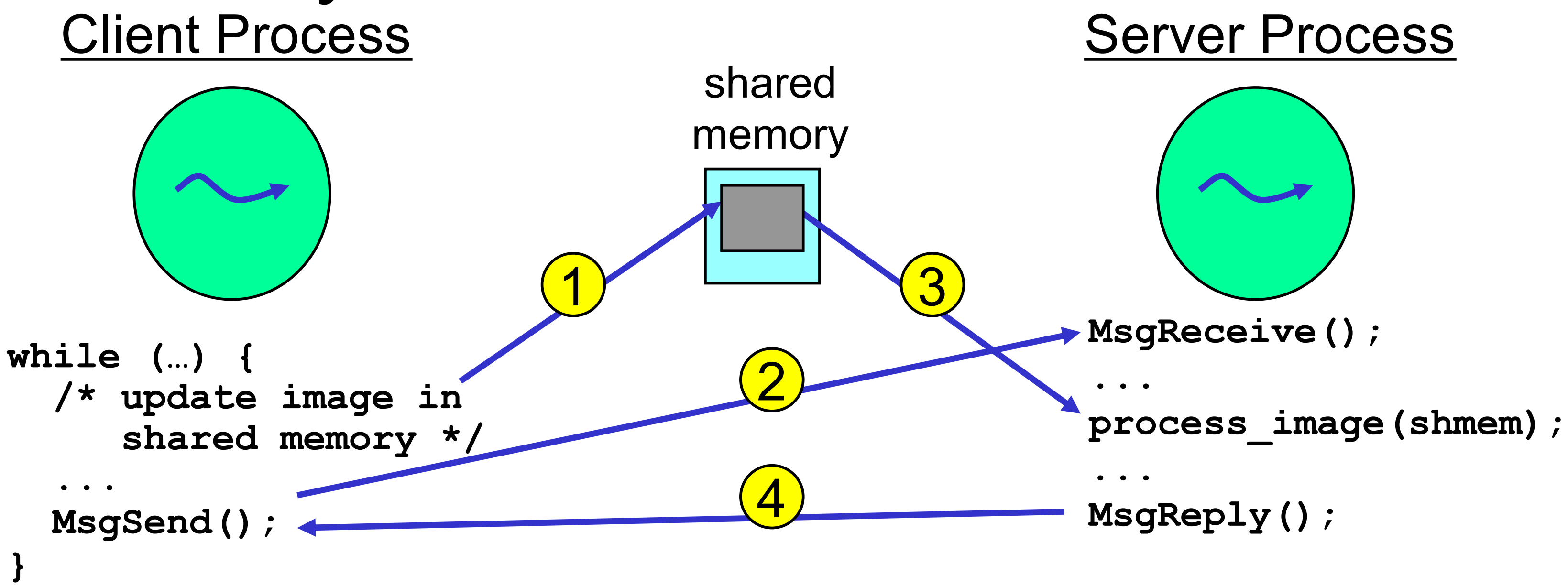
- thread synchronization objects in the shared memory area
  - must be configured for inter-process access:
    - for unnamed semaphores, the **`pshared`** parameter must be non-zero
    - for mutexes and condition variables, the **`PTHREAD_PROCESS_SHARED`** attribute must be set
- *atomic_*()* functions for control variables
- IPC
  - *MsgSend()/MsgReceive()/MsgReply()* has built-in synchronization
  - use the shared memory to avoid large data copies

Shared Memory – Mutex recovery

What if a process dies with a mutex locked?

- use robust mutexes:
  - start recovery when data needed
  - *pthread_mutexattr_setrobust()*
- use mutex events and revival:
  - notification when the mutex goes DEAD
  - start recovery immediately
  - *SyncMutexEvent()* for notification
  - *SyncMutexRevive()* for recovery

IPC for synchronization:



1. client prepares shared memory contents - update animation image
2. client tells server that a new image is ready - and waits for reply
3. server processes the image that is in the shared memory
4. server replies so that client can prepare another image

Since the *MsgSend()* does not return until the server calls *MsgReply()* this synchronises access to the shared memory.

Shared Memory - Access

To solve the access problems, QNX Neutrino supplies:

- anonymous shared memory objects
  - `shm_open(SHM_ANON, O_CREAT… );`
- shared memory handles
  - client requests a shared memory setup from the server
  - server:
    - creates an anonymous shared memory object
    - creates a handle to that object
    - returns the handle to the client
  - client:
    - converts the handle to an fd
    - maps in the shared memory from the fd

The shared memory handle functions:

- handles are opaque objects used for coordinating access to a shared memory object between two processes

```
shm_create_handle( fd, pid, perms, &handle,
  flags );
```

- create a **`handle`** that will allow process **`pid`** to access the shared memory object **`fd`**

```
fd = shm_open_handle( handle, perms);
```

- convert the **`handle`** to an **`fd`** that can be used to map the shared memory object

Exercise:

- in your ipc project are three shared memory examples:
  - POSIX-style named shared memory objects
  - dead mutex recovery example
  - shared memory handles example
- the next few slides detail each example
- try out and look at the source code for the one(s) that interest you

POSIX (portable) example:

- models a one-to-many "global" configuration object with updates
- **`shmem_posix.h`** defines the contents of the shared memory object
- **`shmem_posix_creator.c`** sets up a named shared memory object and updates it
- **`shmem_posix_user.c`** accesses the shared memory object by name and waits for updates
- to run:

```
shmem_posix_creator /myname
shmem_posix_user /myname
```

What happens if you kill them and restart them? How would you fix the problem?

EXERCISE

Dead mutex recovery example:

- **`shmem_mutex_recovery.c`** is a mutex recovery example
  - it registers an event to know if a mutex goes dead
  - it *fork()*s a child that locks the mutex, then *exit()*s.
  - it attempts to revive the mutex when it gets the event

Shared memory handle example:

- **`shmem_qnx.h`** defines the memory sharing protocol
- **`shmem_qnx_server.c`**:
  - receives the messages
  - creates shared memory objects and handles
  - reads and updates the shared memory
  - cleans up when needed:
    - release message
    - disconnect pulse
- **`shmem_qnx_client.c`**:
  - sends the register, update, and cleanup messages
  - maps in the memory from the handle
- this example uses IPC for synchronization

Topics:

- Message Passing
- Designing a Message Passing System (1)
- Pulses
- How a Client Finds a Server
- Client Information Structure
- Server Cleanup
- Multi-Part Messages
- Designing a Message Passing System (2)
- Issues Related to Priorities
- Designing a Message Passing System (3)
- Event Delivery
- Shared Memory
- → Conclusion

Conclusion to QNX Native IPC

In this section, you've learnt:

- the architecture of QNX IPC
- how to:
  - program with QNX Message Passing
  - use the advanced features of QNX IPC
  - design a message passing system