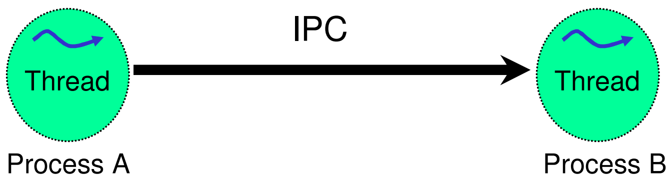
QNX
Inter-Process Communication

QNX SOFTWARE SYSTEMS

NOTES:

IPC:

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



NOTES:

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
    - Persistent Publish and Subscribe (PPS)
  - 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

NOTES:

API: Application Programming Interface

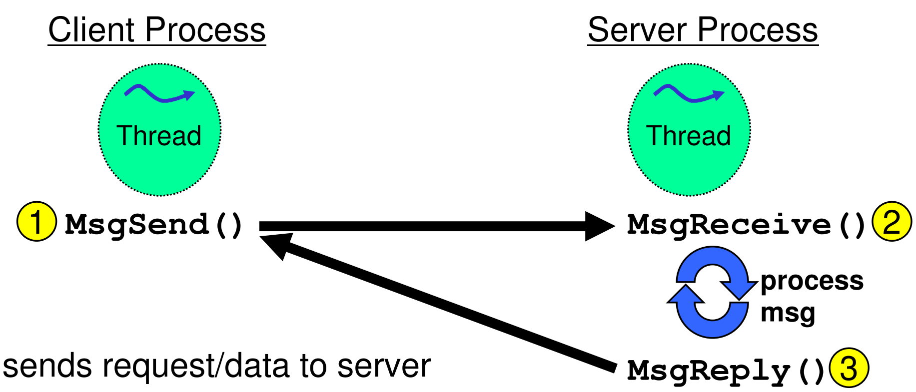
Topics:

- → 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
- QNET
- Shared Memory
- PPS
- Conclusion

NOTES:

QNX Native IPC:

- client-server based
- a bidirectional communication

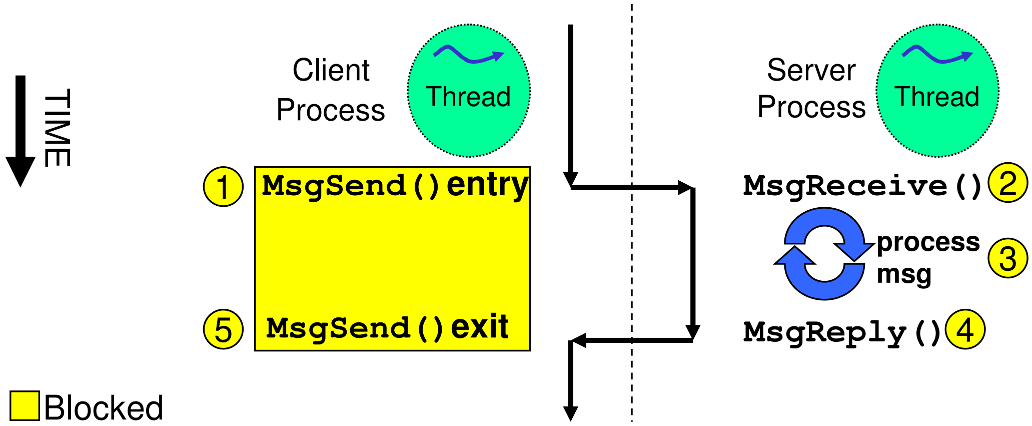


1. client sends request/data to server
2. server receives, and processes
3. server replies to client, client gets response and continues

NOTES:

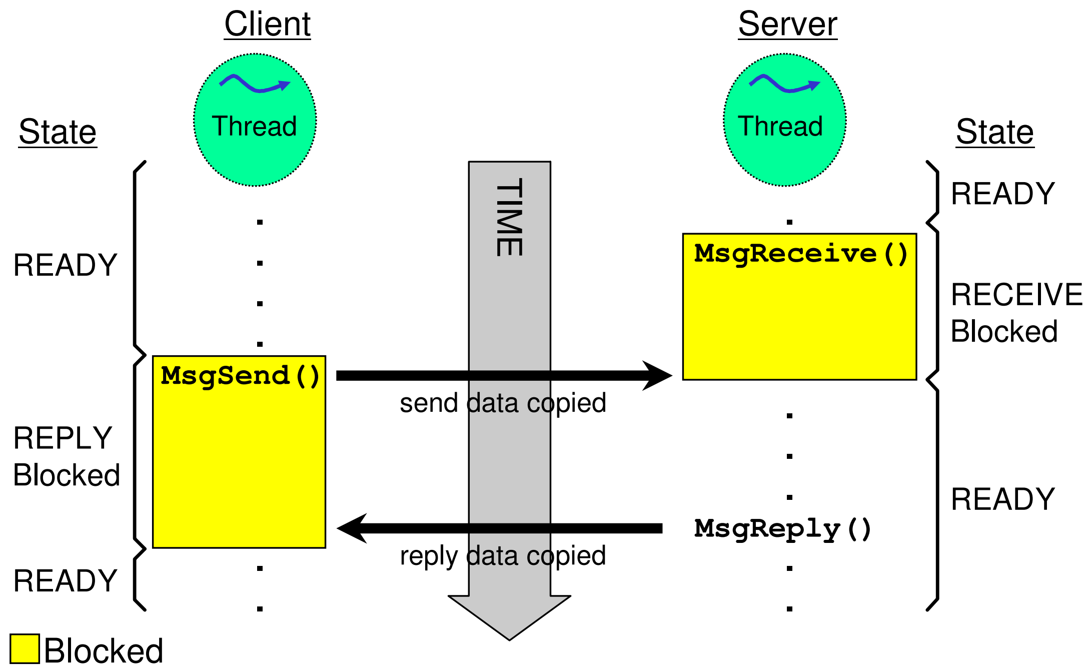
QNX Native IPC is:

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



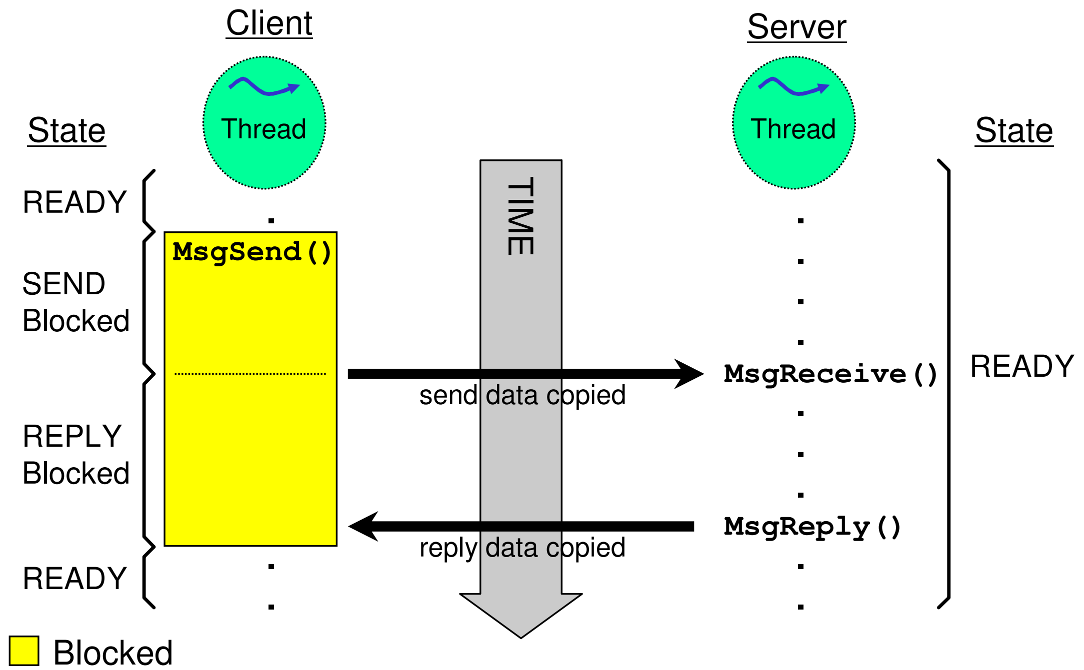
NOTES:

CASE 1: Send after Receive – blocked/idle server



NOTES:

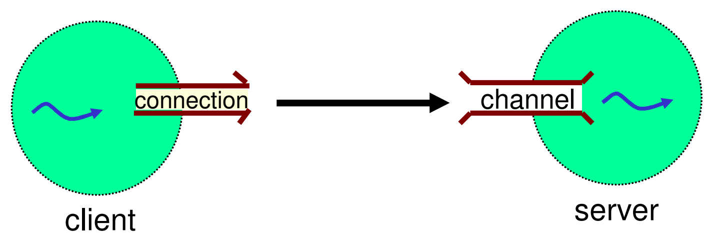
CASE 2: Send before Receive – busy server



NOTES:

Connections and channels:

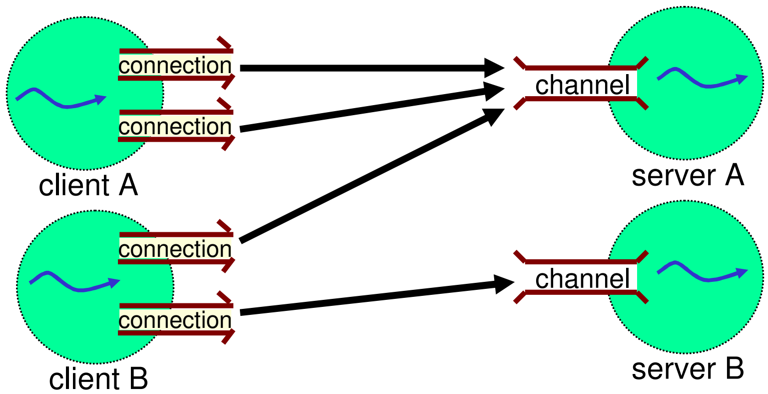
– servers receive on channels,
– clients connect to channels
– a client sends to a server's channel using the connection



NOTES:

Multiple Connection and Channels:

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



NOTES:

Server pseudo-code:

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

Client pseudo-code:

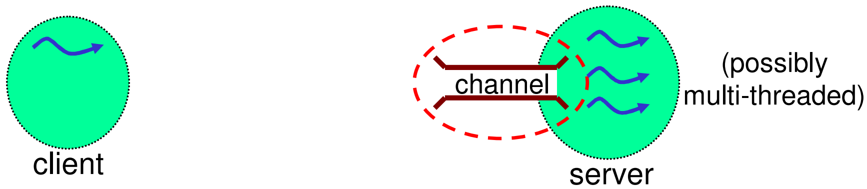
- attach to a channel (*ConnectAttach()*)
- send message (*MsgSend()*)
- make use of reply

NOTES:

In most actual use cases, both *ChannelCreate()* and *ConnectAttach()* are actually covered by another library function, such as *name_attach()* or *resmgr_attach()* for *ChannelCreate()* and *name_open()* or *open()* for *ConnectAttach()*.

The server creates a channel using:

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



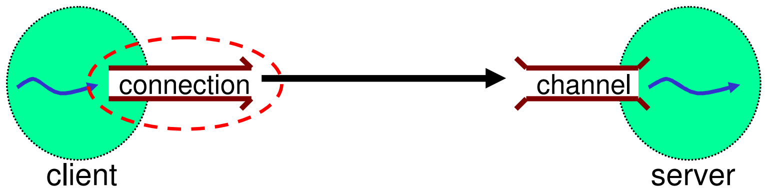
- a channel is attached to a process
- any thread in the process can receive messages from the channel if it wants to
- when a message arrives, the kernel simply picks a *MsgReceive()*ing thread to receive it
- **`flags`** is a bitfield, we will look at some of the flags later

NOTES:

You could have multiple channels associated with a process. Although this is an infrequent situation, this might be used in the case where the process consists of a number of threads, and provides multiple types of services. Clients could connect to one channel for one type of service, and another channel for another type of service.

The client connects to the server's channel:

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



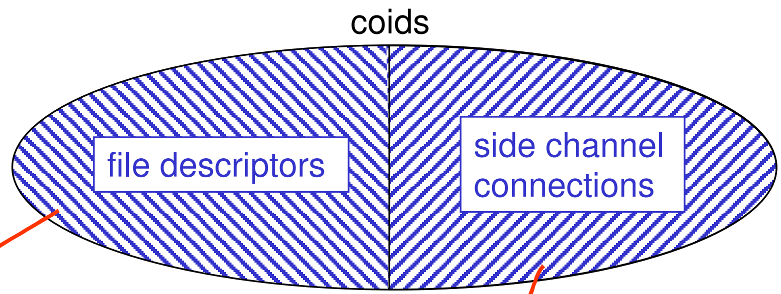
- **nd**, **pid**, **chid** uniquely identify the server's channel
  - **nd** is a node descriptor
    - represents which computer the server is running on
    - use **ND_LOCAL_NODE** if client/server are on same node
  - **pid** is the process id of the server
  - **chid** is the channel id
  - always pass **_NTO_SIDE_CHANNEL** for the 4th parameter (index) -- why? ...

NOTES:

For the node descriptor (nd), use **0** or **ND_LOCAL_NODE** if client/server are on same node. ND_LOCAL_NODE is #defined in <sys/netmgr.h>. In order to determine the nd of the network machine you want to send a message to, you can use **netmgr_strtond()** and other functions.

Message passing over a network relies on the QNET module, which is a plug-in for io-pkt-*, the network manager. We'll look quickly at native networking in the QNET section, and you can learn more about QNET in the System Architecture Guide, and the Programmer's Guide.

Connection IDs (coids) come in two types:




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

coid = ConnectAttach (nd, 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

NOTES:

A missing **_NTO_SIDE_CHANNEL** can have a couple of unfortunate effects:

– a *spawn*()* or *fork()* may deadlock or fail

– the server may receive unexpected messages

In general, you can pass fds to most functions that take a coid, since fds are coids. However, you should not pass a side-channel connection to a call that takes an fd, as the server you connected to usually will not expect the messages generated by the fd based calls.

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

NOTES:

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          10o REPLY        1
   4099   1 proc/boot/pci-bios  12o RECEIVE      1
  20489   1 sbin/pipe           10o SIGWAITINFO
  20489   2 sbin/pipe           10o RECEIVE      1
  20489   3 sbin/pipe           10o RECEIVE      1
  20489   4 sbin/pipe           10o RECEIVE      1
  20489   5 sbin/pipe           10o RECEIVE      1
  69645   1 sbin/enum-devices   10o 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

NOTES:

EXERCISE

Exercise: examine QNX message passing in a system:

- try out the IDE views:
  - Process Information
  - Connection Information
  - System Blocking Graph

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

NOTES:

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

NOTES:

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

NOTES:

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

NOTES:

Don't use a negative value for status because *MsgSend()* already returns -1 on error and *MsgSend_r()* returns a negative **`errno`** value..

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

NOTES:

For those not familiar with **`errno`**, it is a global variable that each thread has its own copy of. In order to return an error status, many functions will set **`errno`** to a value taken from **`<errno.h>`** and then return `-1`. Some (more modern) POSIX functions (e.g. *pthread_mutex_lock()*) will return 0 on success and an **`errno`** value on failure.

Server example

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) );
   }
}
```


NOTES:

This simple skeleton is, almost without exception, at the heart of every server, including resource managers. The only real modifications that occur here are that the *MsgReply()* may be done at a later time -- e.g., after receiving more messages, and that there might be multiple threads blocked on the *MsgReceive()*.

This example also illustrates the fact that the channel is usually created the once, and then used "forever". Most servers don't ever exit.

Client example

The client:

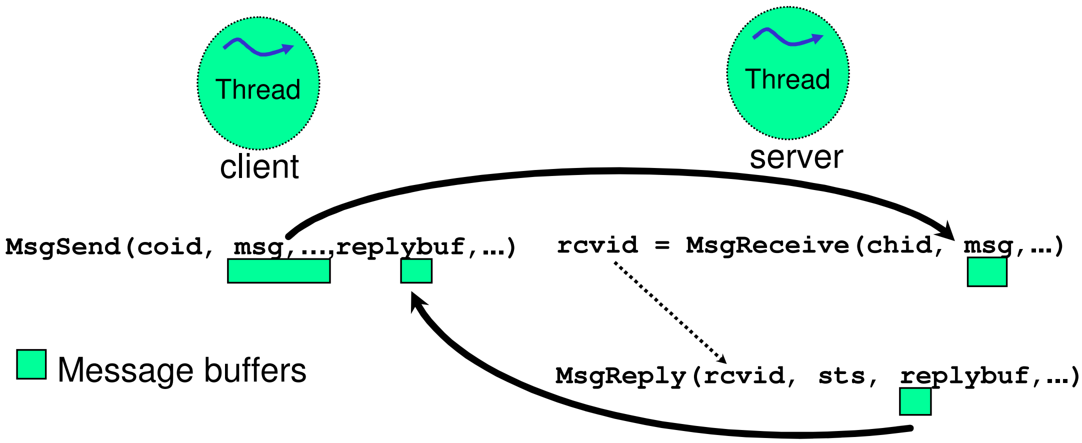
```
#include <sys/neutrino.h>
#include <sys/netmgr.h>

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

   MsgSend(coid, &outgoing_msg, sizeof(outgoing_msg),
                        &incoming_msg, sizeof(incoming_msg) );
   …
}
```


NOTES:

Messaging - The message data

Message data is always copied:

– the kernel does not pass pointers

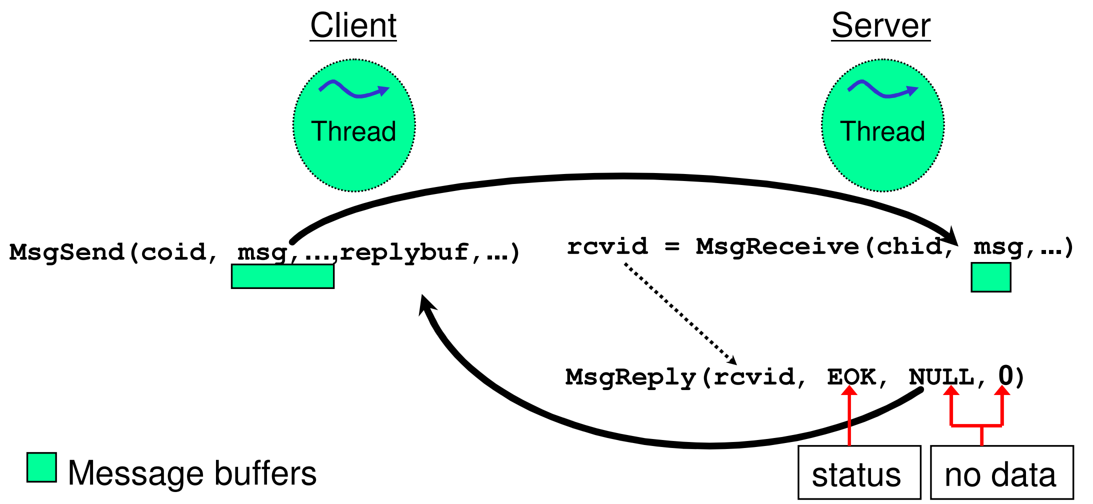


– the number of bytes actually transferred is the smaller of the client buffer size and the server buffer size

NOTES:

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



NOTES:

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...

NOTES:

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

– some questions to consider:

1. what states did the client and server transition through?
   – consider using ‘pidin’ or the IDE’s Process Info. 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?
5. under normal circumstances, the client prints out:
   “**`MsgSend return status: 0`**”
   – where did the ‘0’ come from?

NOTES:

QNX SOFTWARE SYSTEMS

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
- QNET
- Shared Memory
- PPS
- Conclusion

NOTES:

Designing For Message Passing - Message types

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

NOTES:

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

NOTES:

_IO_MAX is defined in **`<sys/iomsg.h>`**

Designing For Message Passing - Server code

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:
     …
   }
}
```


NOTES:

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
- QNET
- Shared Memory
- PPS
- Conclusion

NOTES:

Native QNX Neutrino messaging is inherently synchronous:

- a client's *MsgSend*()* will cause the client to become blocked,
- the server had to do a *MsgReply*()* to unblock the client

What if you didn't want the client to block?

NOTES:

Asynchronous Messaging

There are options:

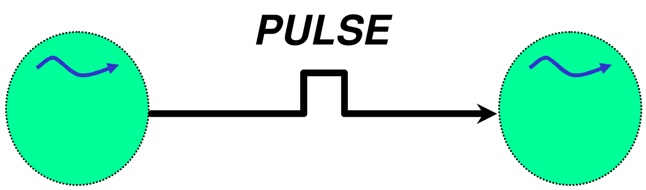
- if you need to transfer data:
  - have the server dedicate a thread
    - the server thread receives the message, and immediately replies, minimizing the blocking interval
  - have the client send the data via a thread
    - effectively a "messenger" thread
  - use a POSIX message queue
- if you don't need to transfer data, or if you simply need to notify somebody that data is available
  - use a POSIX signal
  - use a pulse

Let's look at pulses…

NOTES:

Pulses:

- non-blocking for the sender
- fixed-size payload
  - 32 bit value
  - 8 bit code (-128 to 127)
    - negative values reserved for system use
    - 7-bits available
- unidirectional (no reply)
- fast and inexpensive



NOTES:

Sending Pulses

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`** to **`_PULSE_CODE_MAXAVAIL`**
- **`priority`** indicates what priority the receiving thread should run at
  - QNX uses a priority inheritance scheme to minimize priority inversion problems, as we'll see later
  - delivery order is based on priority
- to send a pulse across process boundaries, the sender must be the same effective userid as the receiver or be the root user

NOTES:

`_PULSE_CODE_MINAVAIL` to **`_PULSE_CODE_MAXAVAIL`** are **`0`** to **`127`** and defined in **`<sys/neutrino.h>`**. Negative pulse values are reserved for system use.

The permission rules for pulses are the same as those for signals.

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 from *MsgReceive()* will be >`0` if a **message** was received.
    - this value will be the **`rcvid`**, which will be needed to *MsgReply()*
  - the return value from *MsgReceive()* 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

NOTES:

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
    }
}
…
```


NOTES:

Pulse Structure

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

```
struct _pulse {
  signed char        code;    <-- 8-bit code
  union sigval       value;
  int                scoid;
};
```

The value member is actually a union:

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

QNX IPC

2010/06/22 R11

NOTES:

This is an OS defined structure, do not try to define your own pulse structure.

Receiving Pulses - Example

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;
```


NOTES:

EXERCISE

Pulse Exercise:

- in your **`ipc`** project
  - copy the checksum **`server.c`** and **`client.c`** from last exercise to **`pulse_server.c`** and **`pulse_client.c`** (respectively)
- extend the server so that it can now receive pulses as well as checksum request messages
- whenever the server receives a pulse, it should print out an indication of this, and it should indicate what the pulse code was
- add some code to **`pulse_client.c`** to send a pulse
  - choose your own pulse code, value, and priority
- run the server, write down, or 'copy' the PID and CHID
- run the client, using the PID, CHID; test out the exchange of messages and pulses
- in the solutions dir/project these are called **`pulse_server.c`** and **`pulse_client.c`**

NOTES:

int getprio(pid_t *pid*) returns the current priority of thread 1 in process *pid*. If *pid* is zero, the priority of the calling thread is returned.

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
- QNET
- Shared Memory
- PPS
- Conclusion

NOTES:

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(nd, pid, chid,…);`
- the problem is, how does the client know what the server's **`nd`**, **`pid`**, and **`chid`** are?

continued...

NOTES:

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`**

NOTES:

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", ... );
```

OR network case:

```
fd = open("/net/nodename/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
```

QNX IPC 2010/06/22 R11

A subsidiary of Research In Motion Limited 45

NOTES:

/net is the default prefix registered by qnet, QNX's native networking protocol. It creates directories for each node under **/net** based on that node's hostname. This prefix can be overridden. To enable native networking you must run the networking stack, **io-pkt**, and load the qnet shared object, **npm-qnet.so**.

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 );
```


NOTES:

The attached names end up in **/dev/name/local** and the components of the path will not be created unless they are needed.

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

NOTES:

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

`_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

NOTES:

We'll look at handling the disconnect and unblock pulses in more detail in the server cleanup section coming up.

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
        …
```


NOTES:

CONNECT messages from *name_open()*

If message passing over a network, *name_open()* may send a CONNECT message to the server:

- for client's *name_open()* to succeed, server must *MsgReply()* to it with a status of EOK, e.g.

```
if(msg.type == _IO_CONNECT) {
      MsgReply(rcvid, EOK, NULL, 0);
}
```

status no data

NOTES:

Note that *name_open()* does not always send a CONNECT message to the server. For example, if message passing locally, and the global name server (**gns**) is not running, no CONNECT message will be sent. Having the above CONNECT message detection code doesn't cause a problem in the local case, but it will allow it to work across network nodes, therefore we recommend that it always be put in.

EXERCISE

Exercise: how a client finds the server

- again, you will be extending the checksum server and client files in your **ipc** project
- copy them (or the pulse versions) to **name_lookup_server.c** and **name_lookup_client.c**
- 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 make use of the 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

NOTES:

In the solutions dir/project, these are called **name_lookup_server.c** and **name_lookup_client.c**.

Topics:

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

NOTES:

Message Information - Getting the information

To get information about the client:

– 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);
```


NOTES:

Message Information - struct _msg_info

The **`_msg_info`** structure contains at least:

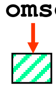
`int`	`nd;`	node descriptor
`pid_t`	`pid;`	sender's process ID
`int`	`tid;`	sender's thread ID
`int`	`chid;`	channel ID
`int`	`scoid;`	server connection ID
`int`	`coid;`	sender's connection ID
`int`	`priority;`	sender's priority
`int`	`msglen;`	# of bytes copied (see next slide…)

NOTES:

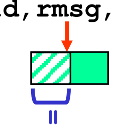
Message Information - Message sizes

Message size info:

```
MsgSend(coid,omsg,100,imsg,300)
```



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



```
info.msglen = 100
```

– 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()*

NOTES:

If calling *MsgInfo()*, **`msglen`** is only valid before the next call to *MsgRead*()* or *MsgWrite*()*.

In the network case, **`msglen`** can be both less than what the client wanted to send, and less than what the server asked for. So you might to do something like:

```
rcvid = MsgReceive(chid, rmsg, rbytes, &info);
if (info.msglen < rmsg.hdr.size - sizeof(rmsg.hdr) )
    MsgRead(rcvid, ...);
```

`rmsg.hdr.size` is the number of bytes in the data portion of the msg. The client would have specified this in the header portion of a multi-part message.

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 clients from a certain node, or a certain process, are allowed to talk to this server
  - *ConnectClientInfo()* can be used to get further information about the client, based on the **`scoid`**
- debugging and logging
  - the server may create usage logs or debug logs, and having the nd, pid and tid logged is useful

NOTES:

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
- …

NOTES:

Maintaining per-client information

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

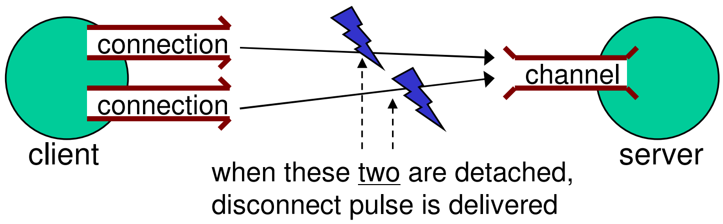
- e.g. client status, requests pending/ongoing
- this type of information must persist as long as the client is connected to the server
- needs to be cleaned up (freed) when client disconnects
- this becomes important for event delivery, as we'll see later

NOTES:

ChannelCreate flags - Disconnect

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
    - loss of network connection if msg passing over a network



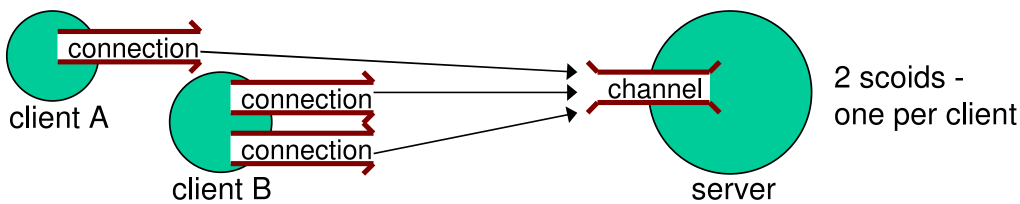
- the pulse **`code`** will be **`_PULSE_CODE_DISCONNECT`**

NOTES:

ChannelCreate flags - 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



NOTES:

You can get the **`scoid`** from the info structure, the 4th parameter to *MsgReceive()* or from the *MsgInfo()* call. We’ll see this later in the Client Information section.

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;
     …
}
```


NOTES:

The manual cleanup (**`ConnectDetach(msg.pulse.scoid)`**) is required as a synchronization between your server and the kernel. If the kernel notified you, and then automatically cleaned up, there would be possible race conditions with the **`scoid`** being re-used before you've done your cleanup. The manual handshake prevents this race condition.

Exercise: 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...

NOTES:

Exercise: cleanup upon client disconnect (continued)

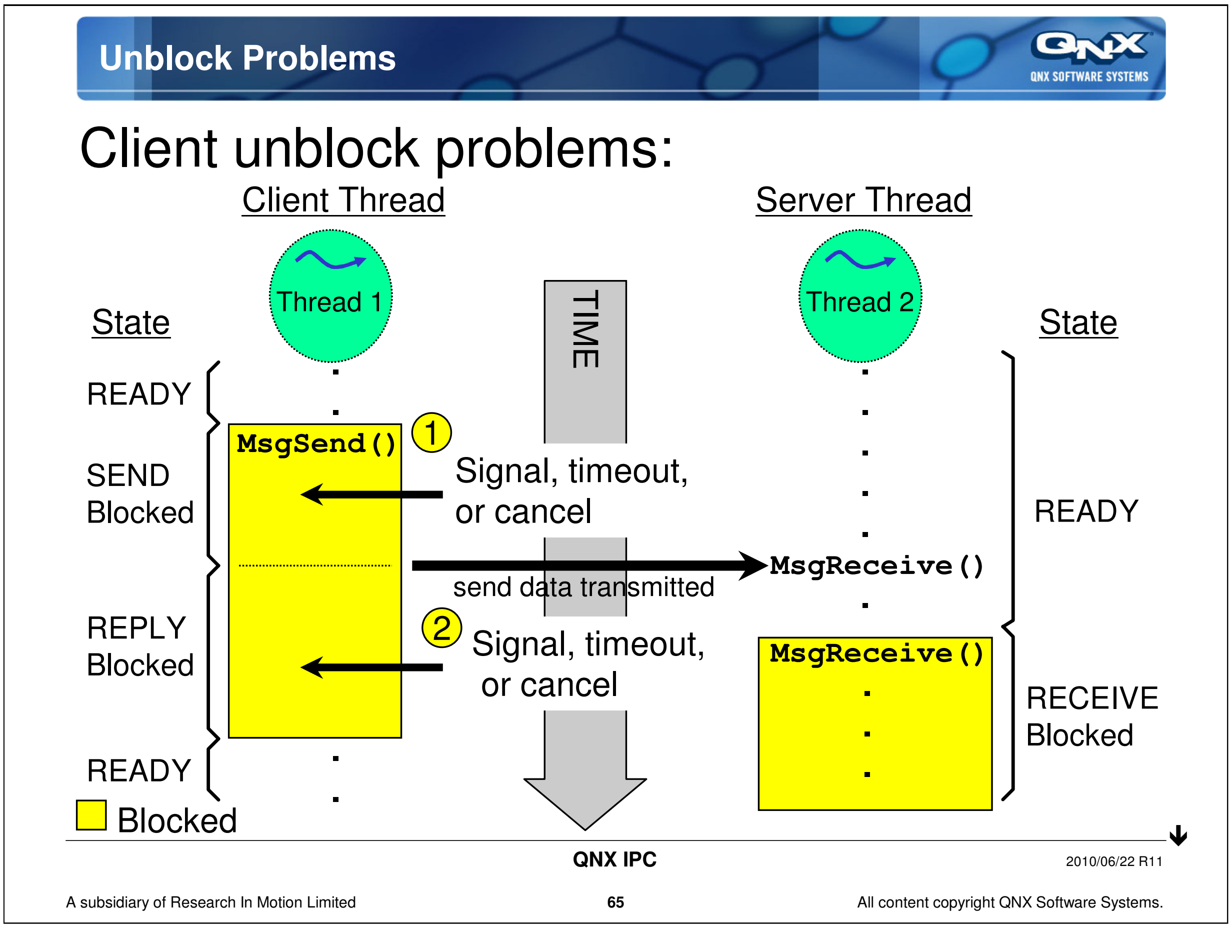
- uncomment the code for the server, so that it cleans up the **`scoid`** and the per-client info every time a client disconnects/terminates
  - *remove_client_from_list(…,scoid);* cleans up the per-client info
  - *ConnectDetach(….scoid);* cleans up the scoid
- rebuild and rerun
  - notice that **`scoid`**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

NOTES:

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
- …

NOTES:



NOTES:

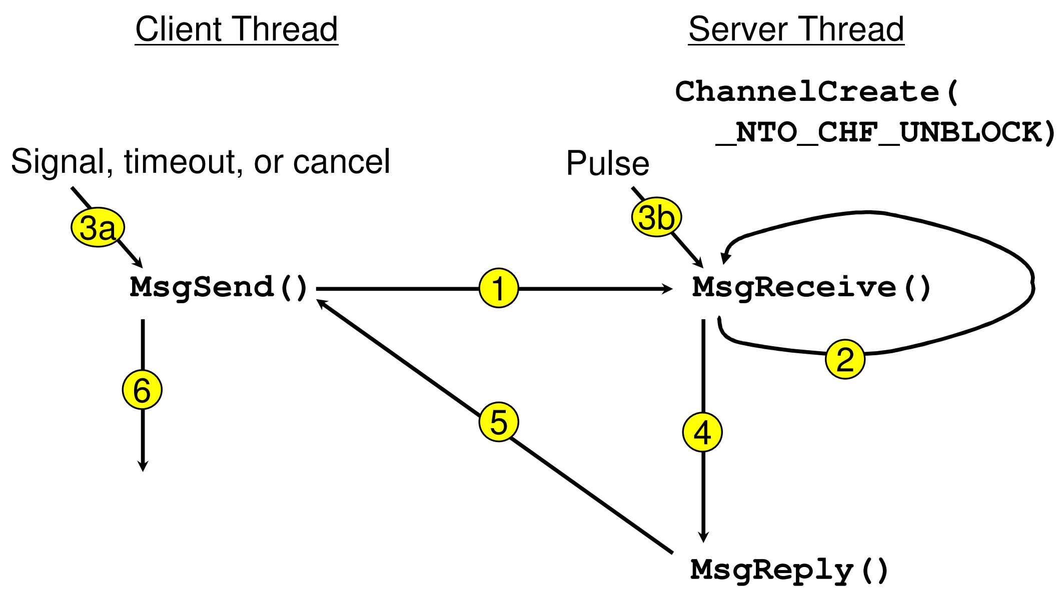
What happens if a client is blocked after sending a message to a server and gets hit by a signal, (or kernel timeout or thread cancellation)?

1) If the signal occurs while the client is SEND blocked, the server has not seen the request, and has not started any operation to handle this request. The send returns -1, the client checks errno for the reason, and retries if needed.

2) But what if the client is REPLY blocked? Now the server has gotten the message and started doing work. Now, if the *MsgSend()* fails, the client unblocks, and if the client retries the operation, the server will get the request twice, which could be bad. Or the server might be doing a long, expensive, operation and the client has terminated from the signal, and can never use the result -- it would be nice if the server could abort the operation.

How can this be resolved?

The unblock flag illustrated:



See the notes for the details.

NOTES:

Notice that it is the server that sets the **_NTO_CHF_UNBLOCK** flag even though if affects both the client and the server.

1. Client sends a message to the server and the server receives it. At this point the client is REPLY blocked.
2. Server receives the message but does not reply. Instead it starts some I/O (for example) and goes back to its *MsgReceive()*. Later when the I/O is ready, an interrupt handler can unblock the *MsgReceive()*.

3a. A signal is set on the client. Because the server had set the **_NTO_CHF_UNBLOCK** flag the kernel makes the signal pending and leaves the sender blocked.

But, this is a bad thing. The signal did not take effect, a potential delay!

3b. So, at the same time, the kernel sends a **_PULSE_CODE_UNBLOCK** pulse message to the server, informing it that the client wants to unblock.

4. The server cancels/cleans up/finishes the I/O and ...
5. ... replies to the client.
6. The client unblocks, any signal handling is done, and the *MsgSend()* returns success.

Or, the server could fail the request and unblock the client with *MsgError()*, and the client's *MsgSend()* would return a failure.

ChannelCreate flags - Unblock

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

NOTES:

Note that all pending unblock pulses for that **`rcvid`** are cleared during the *MsgReply*()*/*MsgError()* processing.

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.

NOTES:

EXERCISE

Exercise: 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

NOTES:

Topics:

NOTES:

Multi-part messaging :

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

part1 (750 KB)

part2 (500 KB)

part3 (1000 KB)

– 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

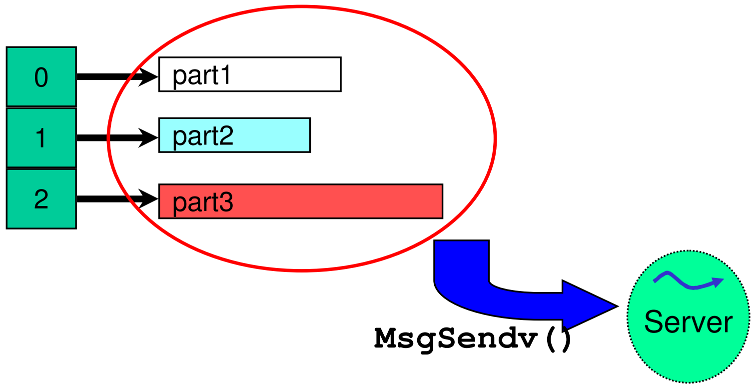
part1 | part2 | part3

continued...

NOTES:

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()*



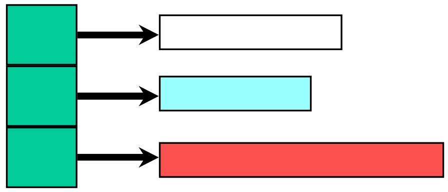
- very useful for scatter/gather situations

NOTES:

Multi-Part Messages - IOVs

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

NOTES:

Messaging - Using IOVs

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

one buffer

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

one buffer

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

array of pointers to multiple buffers

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

array of pointers to multiple buffers

NOTES:

What does an `iov_t` look like?

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

Most useful as an array:

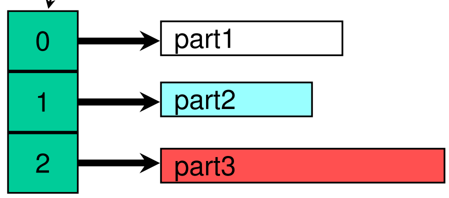
```
iov_t       iovs [3];
```

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

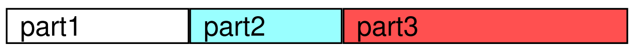
NOTES:

Using an IOV:

each IOV element contains
a pointer and a length




```
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)

NOTES:

The *SETIOV()* macro expands to:

```
#define SETIOV(_iov, _addr, _len)\
((_iov) -> iov_base = (void *)(_addr), (_iov) -> iov_len = (_len))
```

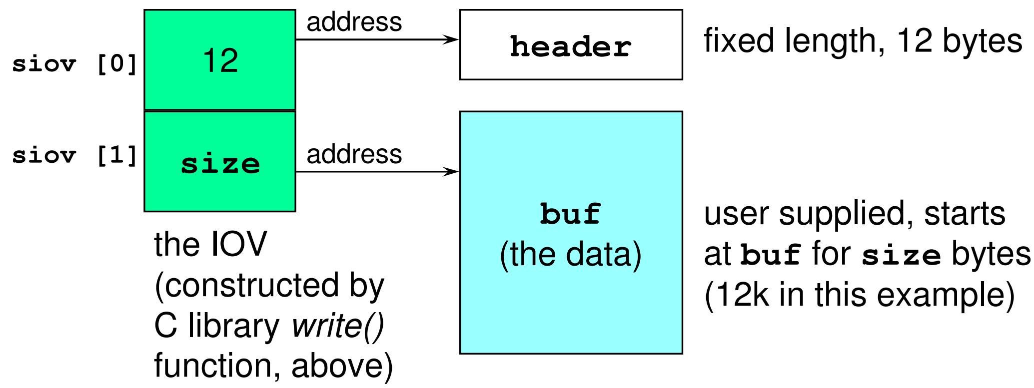
Messaging - Using IOVs

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);
```

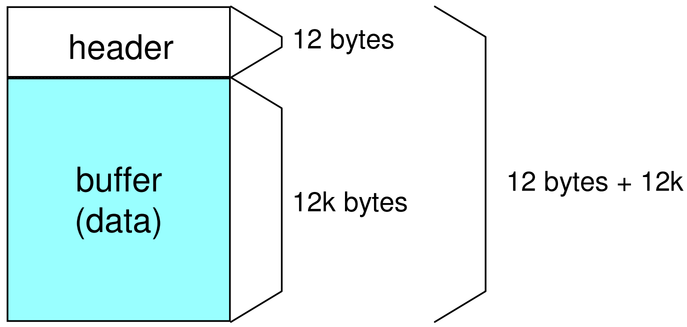


NOTES:

Internally, the C library takes the *write()* request from the user, and attaches a header to the front of it. The entire message is then sent to whichever process is responsible for the device being written to.

Instead of copying the header and the message to an allocated linear buffer, and then sending that, an ***IOV*** is used to inform the kernel that it should use the various pieces from the areas described by the ***IOV***.

What actually gets sent:



A contiguous stream of bytes

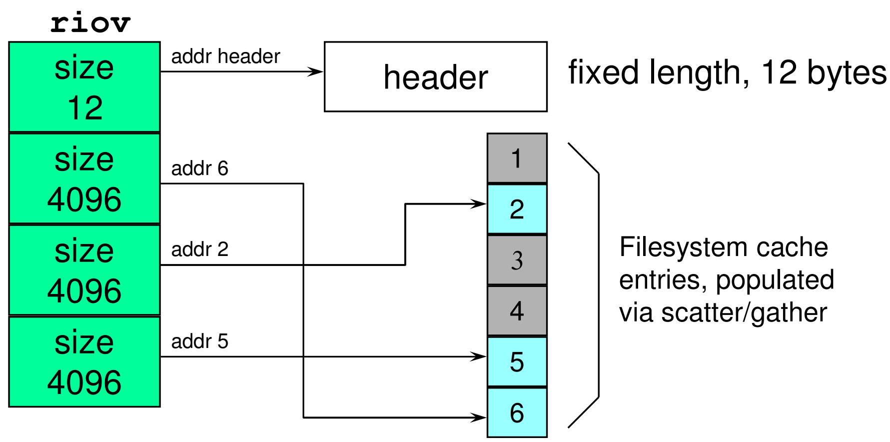
NOTES:

There is no particular “grouping” of the bytes as the kernel copies them from the client to the server -- they can be considered as one contiguous sequence of bytes, with no interruption or “boundaries”.

Messaging - Using IOVs

On the server side, what gets received:

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



NOTES:

Since the kernel didn't enforce any particular grouping of the bytes being copied from the client, the server is free to interpret those bytes in any kind of grouping that makes sense to it, regardless of the client's perceived data organization.

Here we see a filesystem component that has set up an IOV to read the header into a particular structure in memory, and then has broken the message up into 4k chunks and stored them in cache buffers directly -- without the need to copy data!

(A note for the astute reader -- What is shown above assumes that the receiver knows how many bytes will be arriving. The next slides detail a few functions available for the real-world case, where you don't know how many bytes you will be receiving until after you've started to process the message)

Messaging - Using IOVs

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

NOTES:

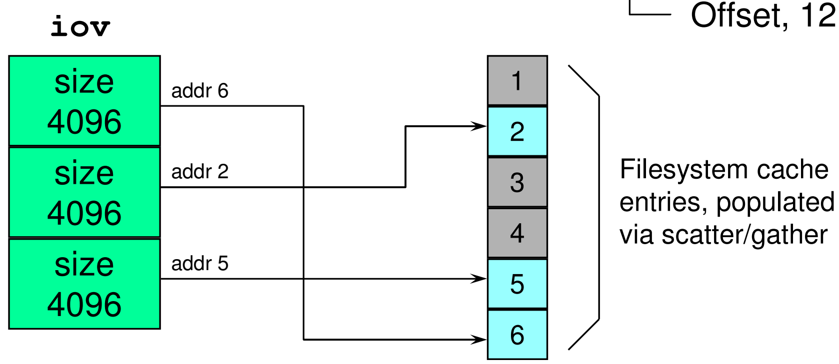
This first call gets just the header. The server software then looks at the header, and constructs an IOV which describes where to put the rest of the bytes -- now that the server knows how many bytes are expected.

Note that the kernel will ***never*** write past the end of the buffer specified in an IOV. There is one small twist, however. When receiving a pulse, the first part of an IOV must be big enough to hold a pulse (**`struct _pulse`**). If it isn't, the *MsgReceive*()* function will return -1 with an **`errno`** of **`EFAULT`** (Bad address.)

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



NOTES:

The *MsgRead*()* call continues the copying process from the client to the server, starting at the specified offset (12 in this case, because the header was 12 bytes long). Since the server has now pre-computed the addresses of the cache buffers, the rest of the message can be received.

Note: The thread being read from must not yet have been replied to, and must be in the **`REPLY_BLOCKED`** state.

You can call *MsgRead*()* as many times as you require to fully process the message.

If it turns out that you don't want to read all of the data provided, there is nothing saying that you have to. Simply ignore the rest of the data, and do the *MsgReply*()* or *MsgError()* to the process.

The *MsgRead()* call:

```
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`**,

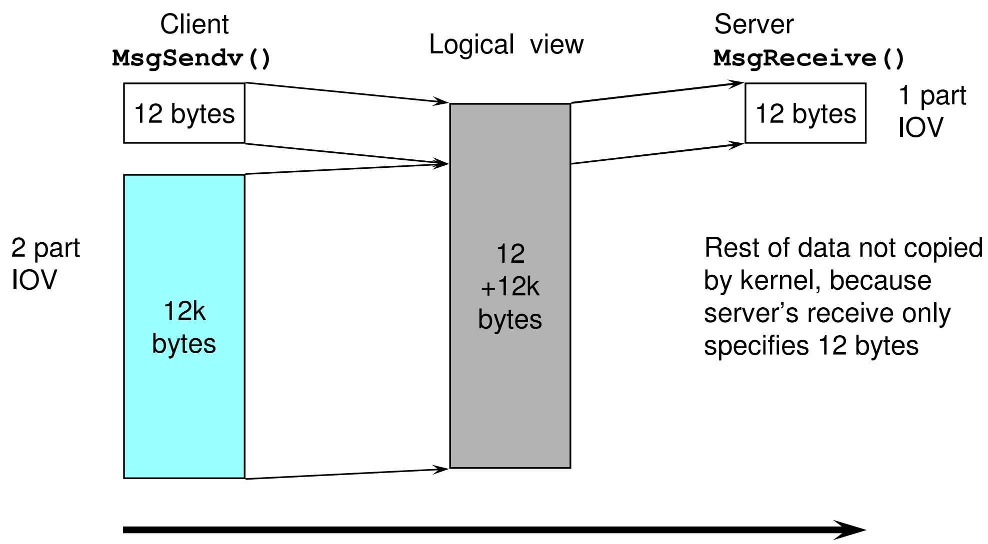
`offset` is the position within the clients send buffer to start reading from

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

NOTES:

Messaging - Using IOVs

From client to server:



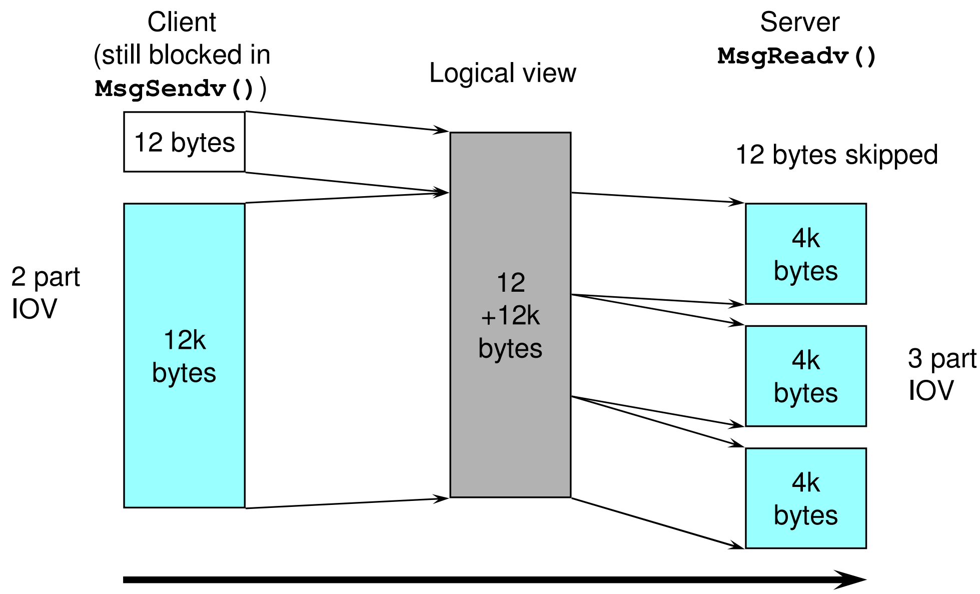
Data copied from client's address space to server's address space by kernel

NOTES:

Note that the kernel doesn't actually keep a copy of the message! The "logical view" is a virtual view, meant only to illustrate that the way a message has been assembled is completely unknown to the server processing the message.

Because the server only requested that 12 bytes be transferred from the client to the server, the kernel only copied the smaller of the two specified sizes -- the 12 bytes. The key point here is that the client is still blocked -- meaning that the server is free to get the data over as many operations as is required. Here we've shown the server getting just the header...

Continuing from client to server:



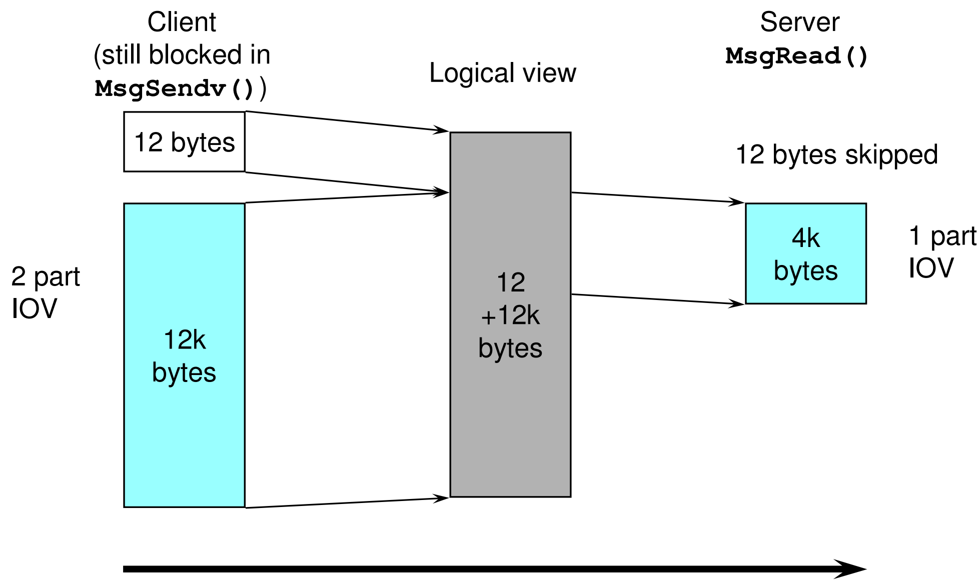
Data copied from client's address space to server's address space by kernel

NOTES:

After the server has analyzed the header, and set up a 3 part IOV to receive the rest of the data, the server then issues a *MsgReadv()* to get more data from the client.
In this example, the server was able to read all 12k from the client in one message transfer.

Messaging - Using IOVs

Or, alternatively:

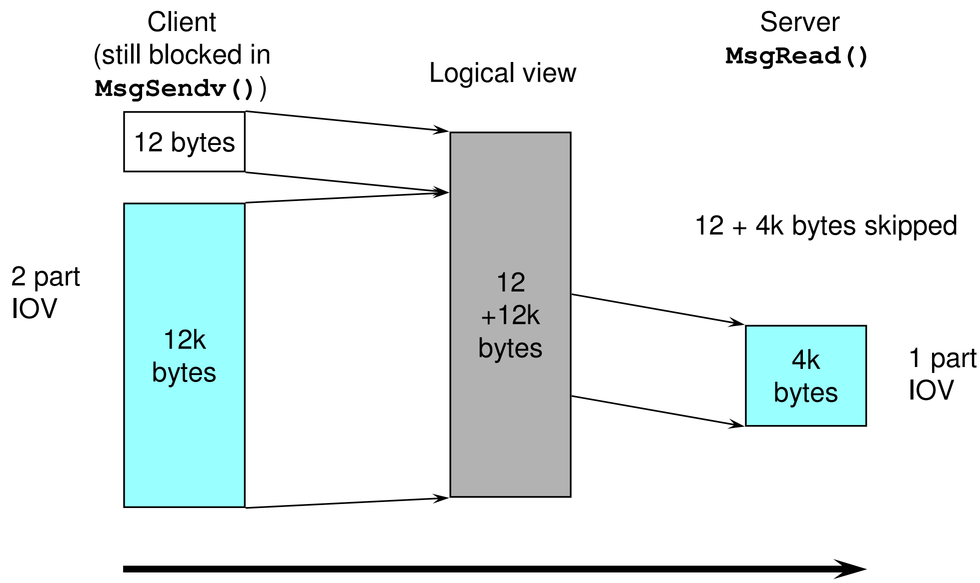


Data copied from client’s address space to server’s address space by kernel

NOTES:

In this alternate example, the server only had 4k of data buffer space available. The server decided that it would read just the first 4k data block from the client, and perhaps read the rest of the data later.

And then getting more later:



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

NOTES:

Then, after the server has received the header and the first 4k message block, it decided that it has room to handle more of the message now. So, the server issues another *MsgRead()* with an offset of (12 + 4k) bytes to continue reading data from the client.

The server is free to re-read data from the client as many times as it deems necessary -- again, since the client is still blocked, we know that the client won't go and modify the data before the server has a chance to get all of it.

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

QNX IPC

2010/06/22 R11

A subsidiary of Research In Motion Limited

87

NOTES:

This is analogous to the *MsgRead*()*. This call too is only for the server's use -- it cannot be used by the client while it is composing the message to the server.

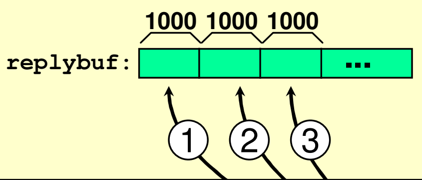
This call would generally be used if the data arrives over time, and is quite large. Rather than buffer all of the data, *MsgWrite*()* can be used to write it into the destination thread's reply message buffer as it arrives.

Another note: *MsgReply*()* must be called to complete the message exchange. But, *MsgReply*()* can be called in one of two ways -- with a message containing data, or an empty message. If it is called with data, that data is written to the FRONT of the destination thread's reply buffer, overwriting any previous contents. This is ideal for writing a header *after* the actual data described in that header has already been written! But, often the status paramater will be used instead.

MsgWrite Example

Example:

```
nbytes = MsgSend(  coid, ...,
                   &replybuf, ... );
```



1. 1st call to *MsgWrite()*
2. 2nd call to *MsgWrite()*
3. 3rd call to *MsgWrite()*

⋮

When no more data, Msg*Reply()*

```
int  offset = 0;
char  data[1000];

while (1) {
  rcvid = MsgReceive( chid,... ); /* for data */
  isData = getSomeData( data );
  while ( isData ) {
    MsgWrite( rcvid, data, 1000, offset );
    offset += 1000; /* set for next MsgWrite()*/
    isData = getSomeData( data );
  }
  /* reply with the number of bytes written
     as the status */
  MsgReply( rcvid, offset, NULL, 0 );
}
```


NOTES:

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);
```


NOTES:

Here you see the big picture. The client specifies two message buffers, a transmit buffer (**`txmsg`**/**`txbytes`**) and a receive buffer (**`rxmsg`**/**`rxbytes`**). The client becomes blocked when it issues the *MsgSend()*, and remains blocked until the server issues the *MsgReply()*.

Note the directions of data transfer. The server's *MsgReceive()* (or *MsgRead()*) transfer data from the client to the server, and the *MsgWrite()* (or *MsgReply()*) transfer data from the server to the client.

It is important to realize that the client ***can*** provide two distinct buffers, **`txmsg`** and **`rxmsg`**. Often, however, the client may choose to provide just ***one*** buffer (**`txrxmsg`**?) for both transmitting from and receiving into. The impact of this is that the server may be overwriting data areas in the client that the server itself may be re-reading from.

One more key point -- the server may choose to handle the client's request piecemeal, like we saw before, in one of two ways. It may choose to read the client's requests in steps, and then, when the entire request has been read, it may choose to write the client's response (again, in steps, or in a single transfer). However, the server may also choose to read a chunk of the client's request, process it, write part of the reply, and go back and read more from the client, process it, write it, and so on.

EXERCISE – Option 1

IOV messaging example:

- examine and run iov_server.c and iov_client.c in your **`ipc`** project
- client will get a string from the user, which can vary in length
  - it creates a 2-part IOV message
    - header that says how many bytes are in the string
    - a buffer that contains the actual data, it simply follows the header
- server 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

NOTES:

EXERCISE – Option 2

IOV messaging exercise:

- extended your **`name_lookup_client`** and **`name_lookup_server`** files in your **`ipc`** project to use IOV messaging
- copy them to **`iov_client_ex.c`** and **`iov_server_ex.c`**
- 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
    - 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

NOTES:

Topics:

NOTES:

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

NOTES:

QNX SOFTWARE SYSTEMS

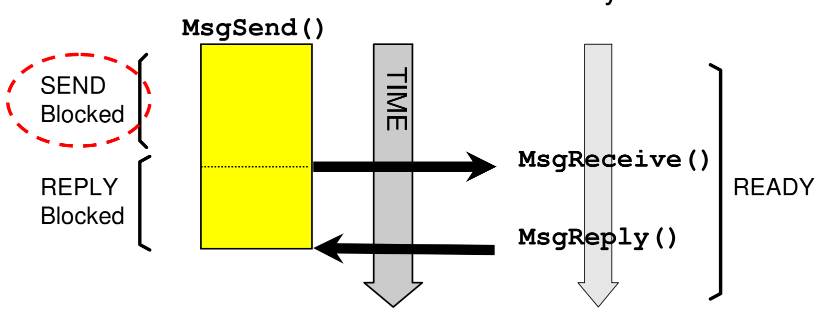
Topics:

NOTES:

If the server calls *MsgReceive()* and there are several clients SEND blocked:

- the message from the highest priority SEND blocked client is received
- if multiple clients have the same priority, the one that has been waiting longest is handled
- this follows the same rules as scheduling

CASE 2: Send before Receive – busy server



NOTES:

If threads call MsgSend() in order:

Thread id	Priority
1	10
2	15
3	10
4	20

They will be received in order: 4,2,1,3

NOTES:

Pulses - Receive order

If a process is receiving messages and pulses:

- receive order is still based on priority, using the pulse’s priority
- receiving thread will run at the priority of the pulse
- pulses and messages may get intermixed

NOTES:

Receive Order - example

QNX SOFTWARE SYSTEMS

If threads send pulses and messages as follows:

TIME

Thread id	Thread priority	Action	
1	10	Send pulse p1 with pulse priority 15	2
2	16	Send message	1
1	10	Send pulse p2 with pulse priority 9	5
3	11	Send message	3
1	10	Send message	4
4	17	Send pulse p3 with pulse priority 6	6

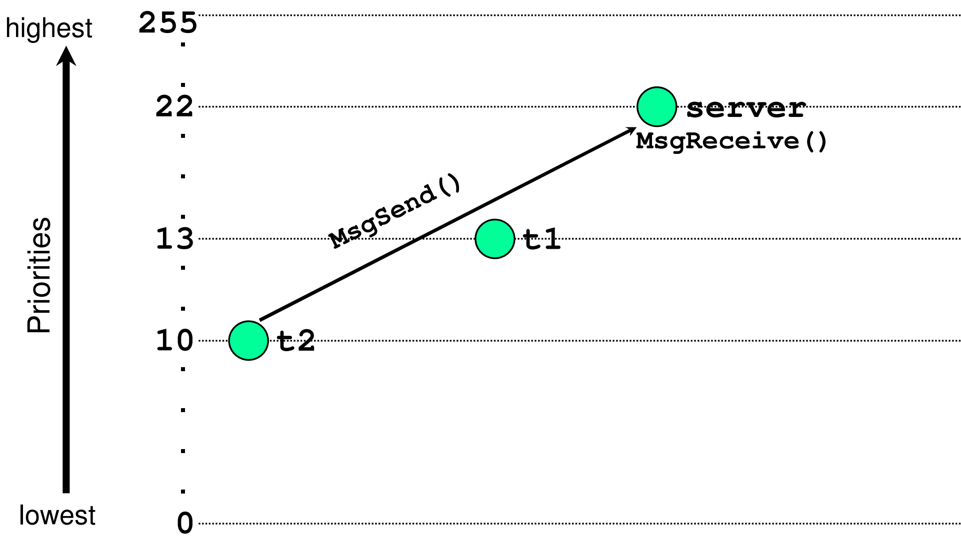
☞ the message from thread 1 gets to the server before the pulse that was sent before it

NOTES:

Priority Inversion - The Problem

QNX SOFTWARE SYSTEMS

The priority inversion problem:



Note: this isn't what really happens, as the next few pages show

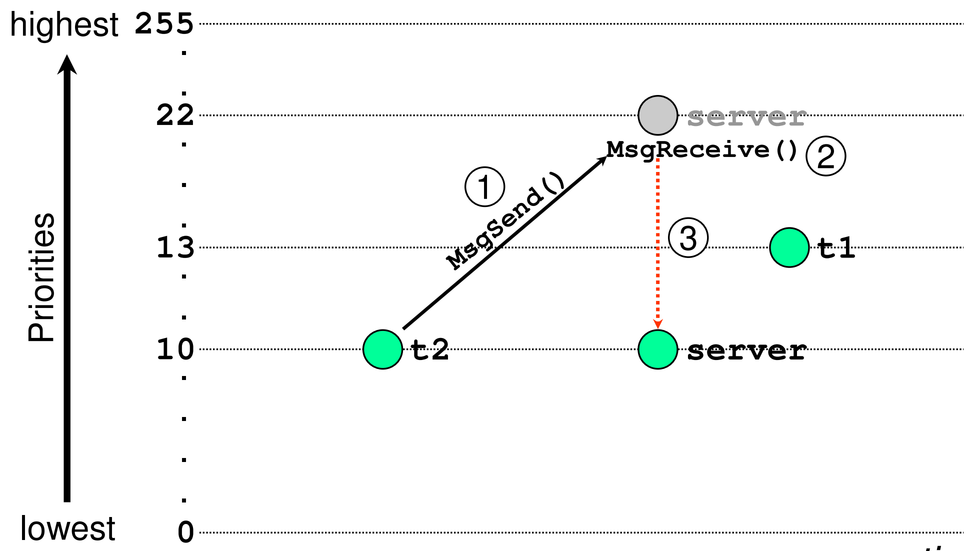
NOTES:

In the diagram above, **`t2`** has sent a message to **`server`**. The work for this message may take a long time. Since **`server`** is now doing **`t2`**'s work, **`t2`** is now effectively running at **`server`**'s priority, 22. We say that **`t2`**'s priority has been inverted.

This also means that **`t2`** is effectively preventing **`t1`**, a higher priority thread, from getting the CPU when it becomes READY. That is the real problem.

Priority Inversion - The Solution

Priority inversion - solution is priority inheritance:



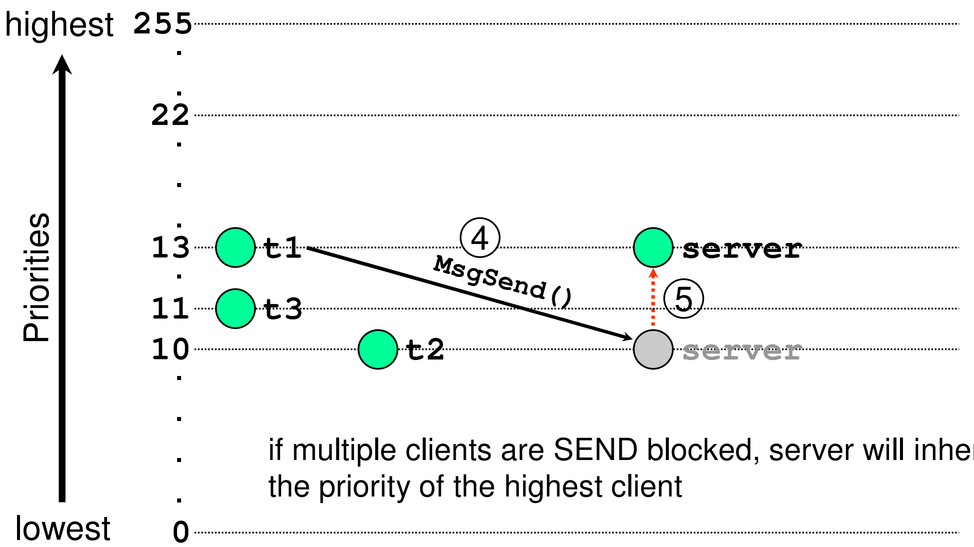
continued...

NOTES:

1. **`t2`** sends to **`server`**.
2. **`server`** receives the message.
3. **`server`**'s priority is dropped to that of the highest priority sender, which in this case is **`t2`**. Note that in this case the sender's priority was lower than **`server`**'s so priority change was done on the *MsgReceive()* (not the *MsgSend()*).

So, problem solved. **`t2`**'s work is being done at **`t2`**'s priority, 10. This time **`t2`** did not effectively preempt **`t1`**.

Solution: (continued)



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

NOTES:

4. **`t1`** sends to **`server`**.

5. **`t1`** is the new highest priority sender to **`server`**. So **`server`**'s priority is raised to **`t1`**'s priority, 13. Note that in this case the sender was at a higher priority than **`server`** so the priority change was done on the *MsgSend()*. In fact **`server`** may still have been working on **`t2`**'s message and known nothing at all about **`t1`**. At least **`t1`** is not waiting for a lower priority thread, **`t2`**.

Note that when **`server`** does finally reply to **`t1`** and **`t2`**, its priority will still be 13. The **`server`** typically next goes back to its *MsgReceive()* and blocks so its priority at this point is irrelevant.

This is analogous to the situation when a higher priority thread tries to lock a mutex owned by a lower priority thread.

Even if **`server`** may not *MsgReceive()* **`t1`**'s message right away, it is still important that **`server`**'s priority is bumped up. Why? What if **`t3`** becomes ready at priority 11? If **`server`** was still down at 10, then it would be waiting for **`t3`** to let it have some time to run so that it could eventually receive **`t1`**'s message. So effectively, **`t1`** would be held up by **`t3`**, a lower priority thread.

This is the default behaviour. You can turn off priority inheritance by creating the channel with the **`_NTO_CHF_FIXED_PRIORITY`** flag.

Topics:

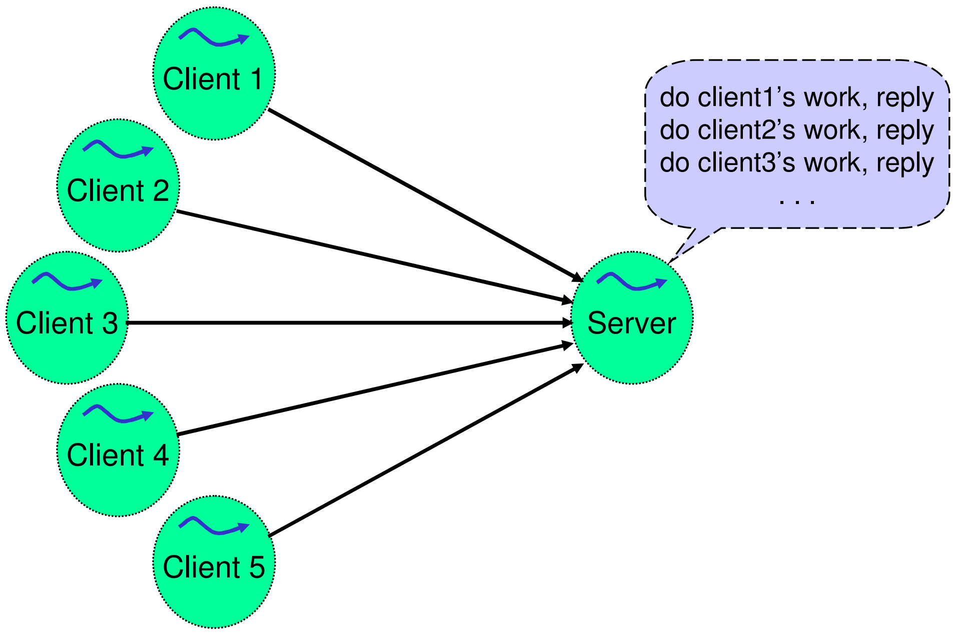
- Message Passing
- Designing a Message Passing System (1)
- Pulses
- How a Client Finds a Server
- Client Information Structure
- Server Cleanup
- Multi-Part Messages
- Designing a Message Passing System (2)
- Issues Related to Priorities
- Designing a Message Passing System (3)
  - Server Designs
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

…

NOTES:

Singlethread Model

Typical server:

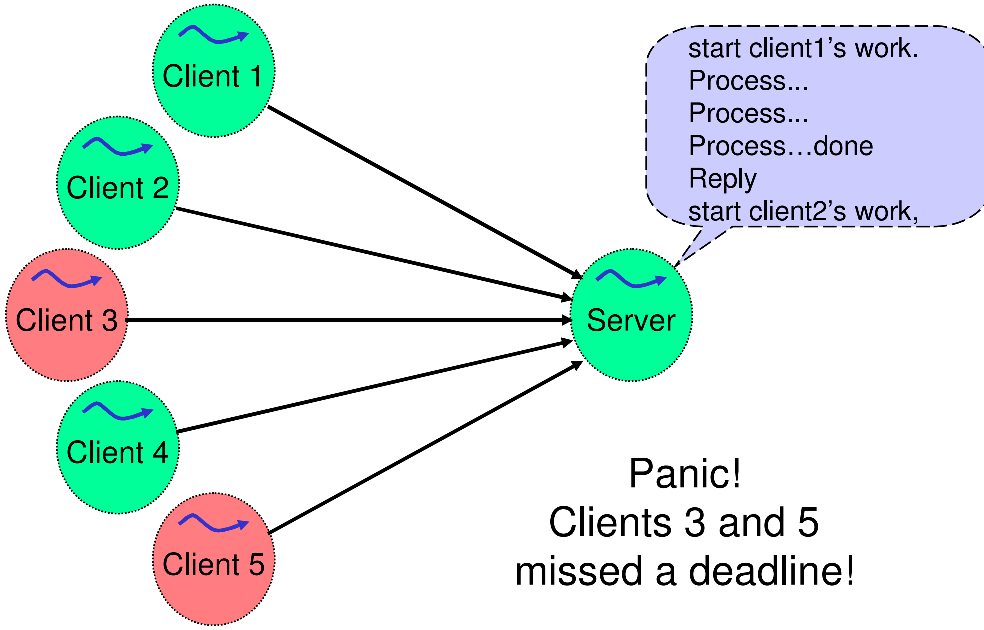


NOTES:

This model is most often used during development, as it provides a very simple debugging environment -- one message is processed to completion, and replied to, and then the next message, etc. Some servers will always use this simple model, especially if the requests can be satisfied quickly.

Singlethread Model

If the request takes a long time:



Panic!
Clients 3 and 5
missed a deadline!

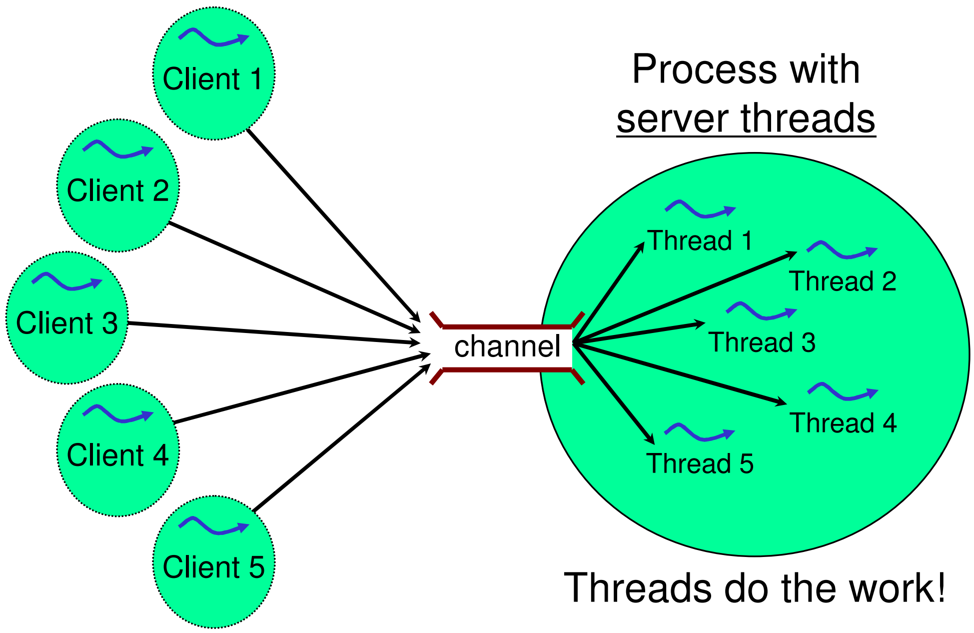
NOTES:

The model that’s usually used to overcome this in a single-threaded server is that the server receives a message, processes it for a while, and then looks around to see if there is another message. If there is, and it is a higher priority, the server switches to the higher priority message and processes it, all the while looking to see if there are other, higher priority messages arriving.

Effectively, the server is doing multi-threading! The kernel is much better at this sort of thing...

Multithread Model

The server can start up some threads:



Threads do the work!

NOTES:

The server starts up a number of threads which all then perform a *MsgReceive*()*. Note that they all ask to receive on the same channel. When a message arrives, a thread is already available to perform the work for a given client:

```
int chid;
main ()
{
  chid = ChannelCreate (…);
  for (i = 0; i < NumServerThreads; i++) {
    pthread_create (… server_thread …);
  }
}

server_thread ()
{
  while(1)
  {
    rcvid = MsgReceivev (chid, …);
    // process
  }
}
```

Multithread Model

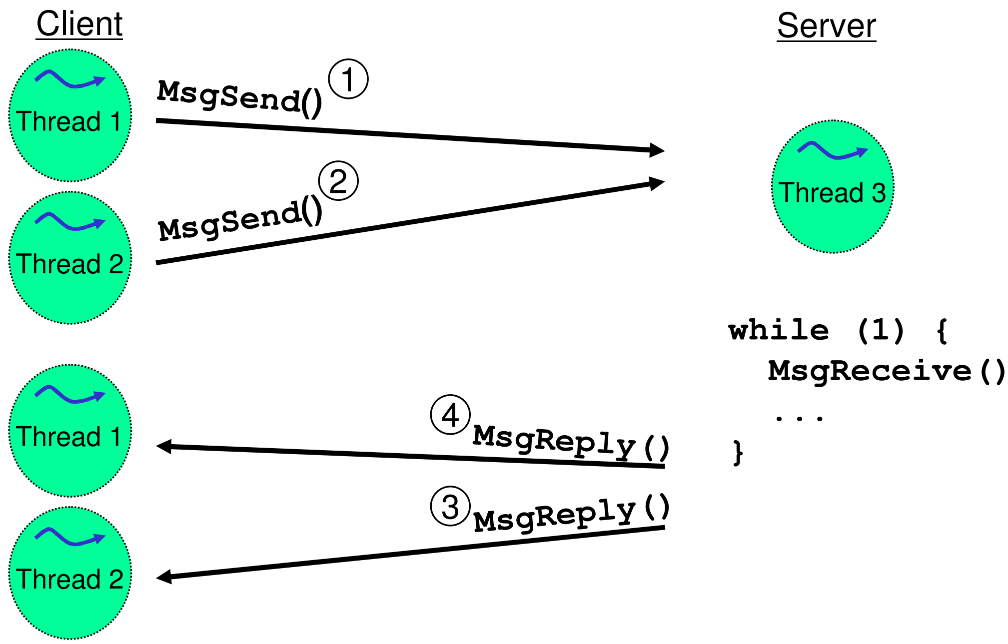
The multithread model:

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

NOTES:

Messaging - Delayed reply

The server doesn't need to reply immediately:



NOTES:

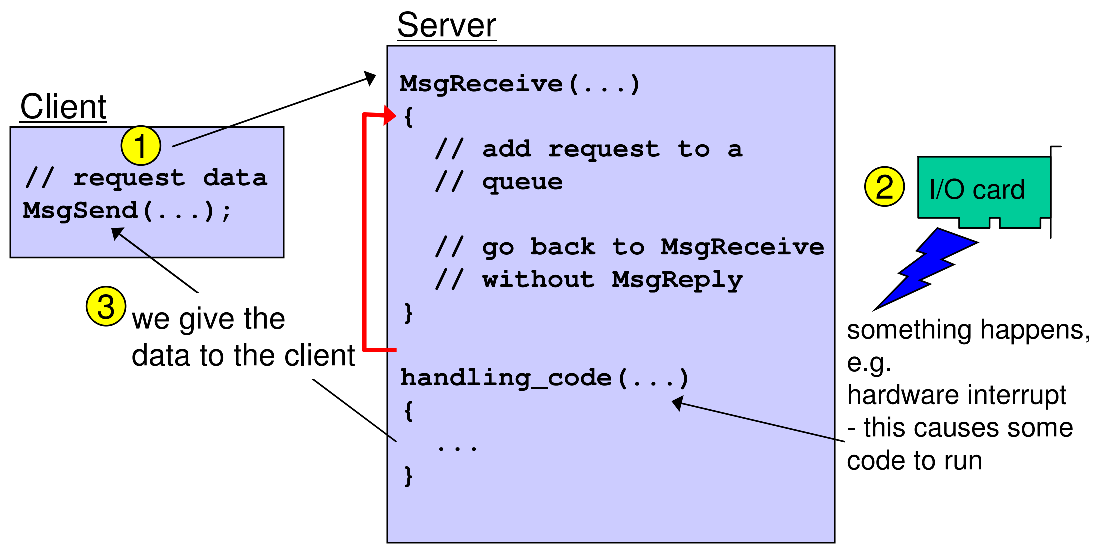
In the above example, the server receives both clients' messages before replying to either one.

1. Thread 1 sends a message to thread 3, which thread 3 receives.
2. Thread 2 sends a message to thread 3, which thread 3 receives.
3. Thread 3 replies to thread 2.
4. Thread 3 replies to thread 1.

The replies could also be done in the order (4) then (3).

If delaying the reply, at minimum the server must store away the rcvid for the client. In general, it will need to store away more, generally this being information about what the client is waiting for.

Delayed reply example:



NOTES:

If delaying the reply, at minimum the server must store away the rcvid for the client. In general, it will need to store away more, generally this being information about what the client is waiting for.

Delayed reply example:

```
    pid tid name                  prio STATE       Blocked
      1   1 /boot/sys/procnto      0f READY
      1   2 /boot/sys/procnto     10r RECEIVE      1
      1   3 /boot/sys/procnto     15r RECEIVE      1
      1   4 /boot/sys/procnto     10r RECEIVE      1
      1   5 /boot/sys/procnto     15r RECEIVE      1
      1   6 /boot/sys/procnto      6r NANOSLEEP
      1   7 /boot/sys/procnto     10r RUNNING
   4100   1 /tinit/x86/o/tinit    10o REPLY        1
  94219   1 devc-pty              20o RECEIVE      1
 450595   1 bin/sh                10o SIGSUSPEND
 573478   1 /pidin/x86/o/pidin    10o REPLY        1
 536617   1 r/photon/bin/pterm    10o RECEIVE      1
 540714   1 bin/sh                10o REPLY        94219
```

shell is blocked, waiting for a reply from pid 94219

pid 94219 (the server) is in the RECEIVE state, waiting for another message, without having replied to the shell

NOTES:

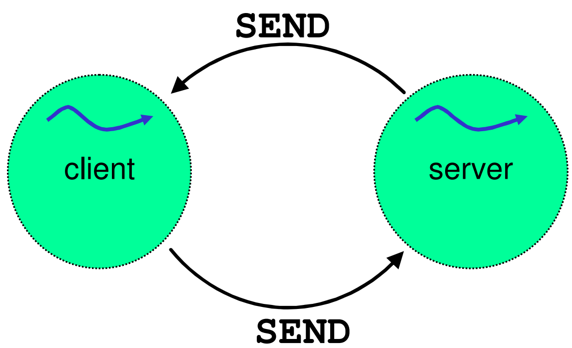
Topics:

- 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
- ...

NOTES:

Designing with messages - Deadlock avoidance

What happens if you need a server to be able to initiate a SEND to a client?

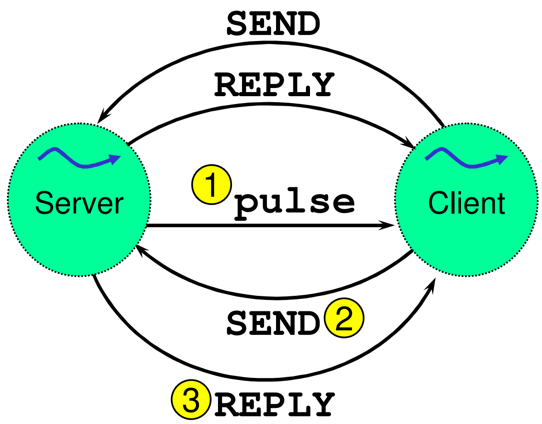


- it's possible to put a channel in a client, but…
- if two processes SEND to each other, they will be blocked, waiting on each other's reply
  - this is a "DEADLOCK"

NOTES:

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



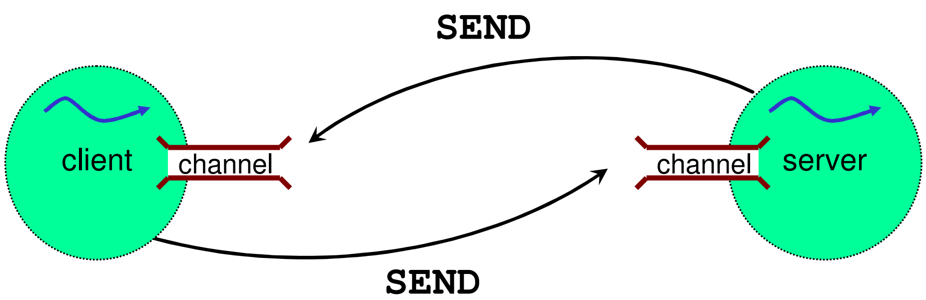
NOTES:

For the client to send to the server, it just sends.

For the server to send to the client:

1. The server sends a pulse to the client. The server then goes back to its *MsgReceive*()*.
2. The client receives the server's pulse and responds by sending a message asking for the data.
3. The server replies with the data.

A client can have a channel:

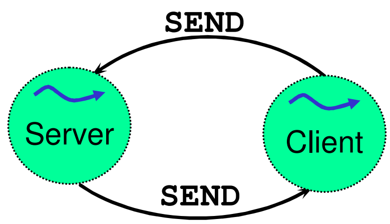


NOTES:

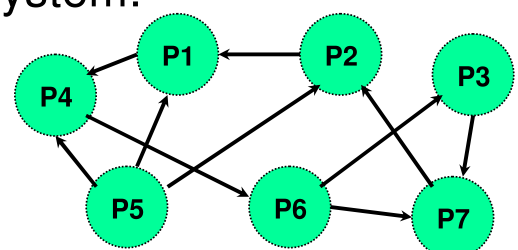
Designing with Messages - Deadlock avoidance

QNX SOFTWARE SYSTEMS

If you only have two processes:



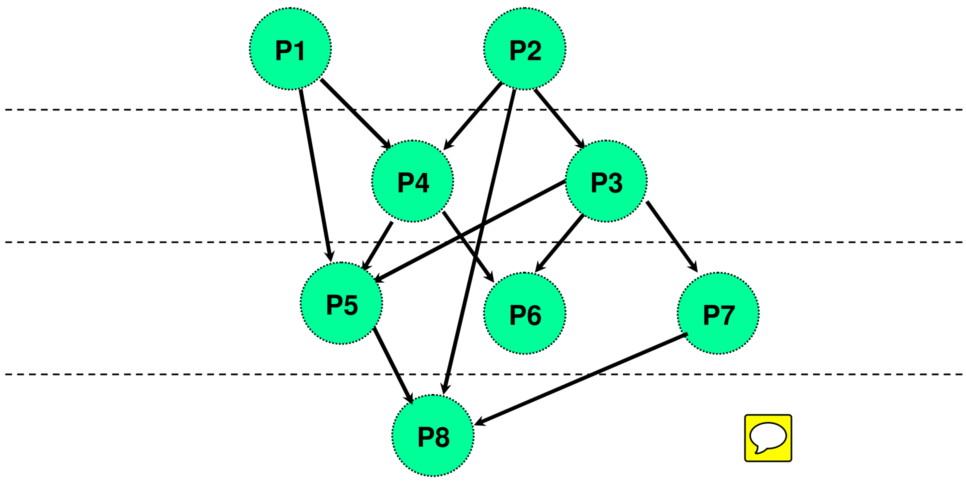
recognizing the potential for deadlock is easy, but in a complex system:



It is much more difficult...

NOTES:

Solution: a Send hierarchy

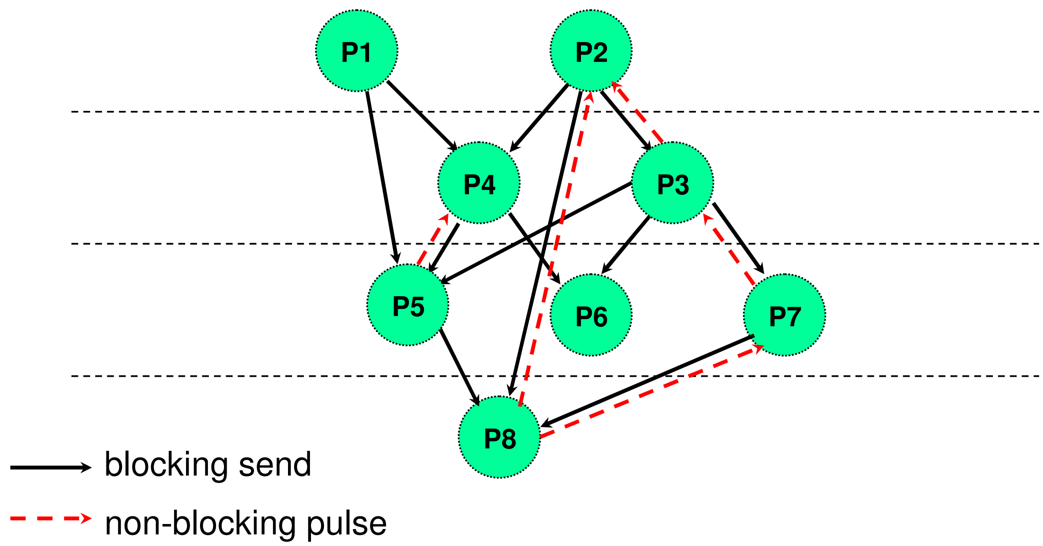


- sends always go downwards so there can never be a deadlock
- if two processes on the same level need to communicate, just create another level

NOTES:

QNX SOFTWARE SYSTEMS

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



NOTES:

Topics:

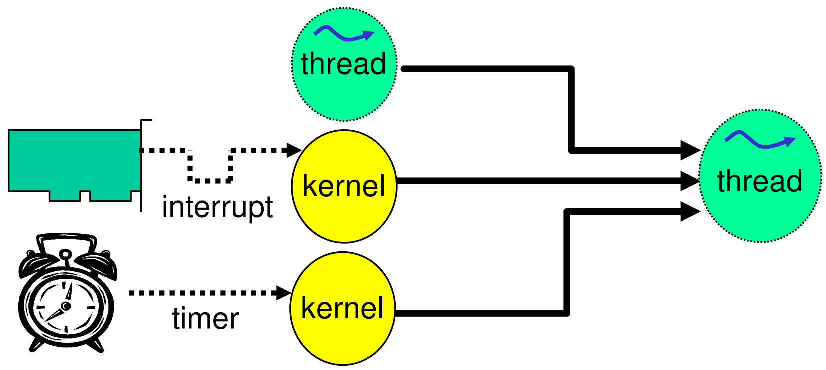
- Message Passing
- Designing a Message Passing System (1)
- Pulses
- How a Client Finds a Server
- Client Information Structure
- Server Cleanup
- Multi-Part Messages
- Designing a Message Passing System (2)
- Issues Related to Priorities
- Designing a Message Passing System (3)
- → Event Delivery
-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

NOTES:

Event Delivery

Events are a form of notification:

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



NOTES:

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

NOTES:

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

NOTES:

Example of manually initializing a pulse event:

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


sigevent.sigev_notify = SIGEV_PULSE;     //we want a pulse event
sigevent.sigev_coid = ConnectAttach (0, 0, chid, ...); // to self
sigevent.sigev_priority = MyPriority;    // our priority
sigevent.sigev_code = OUR_CODE;          // our PULSE ID
sigevent.sigev_value.sival_int = value;  // 32bits of data
```

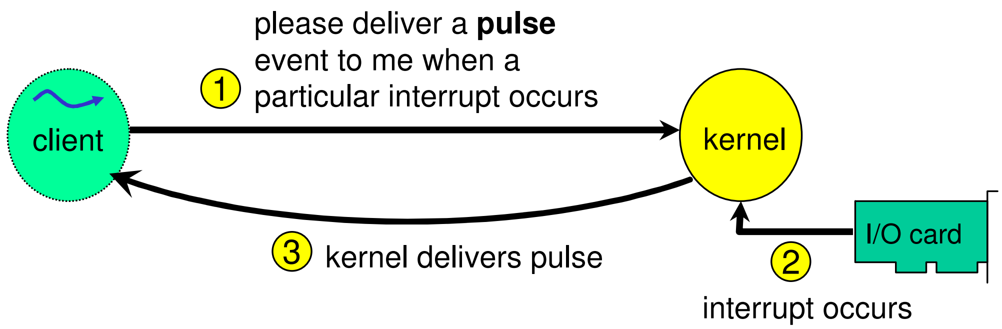

NOTES:

Example of using a macro to initialize a pulse event:

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

//connection to our channel
coid = ConnectAttach (0, 0, chid, _NTO_SIDE_CHANNEL, flags);
SIGEV_PULSE_INIT(&sigevent, coid, MyPriority, OUR_CODE, value);
```


NOTES:

Interrupt and Timer Events:

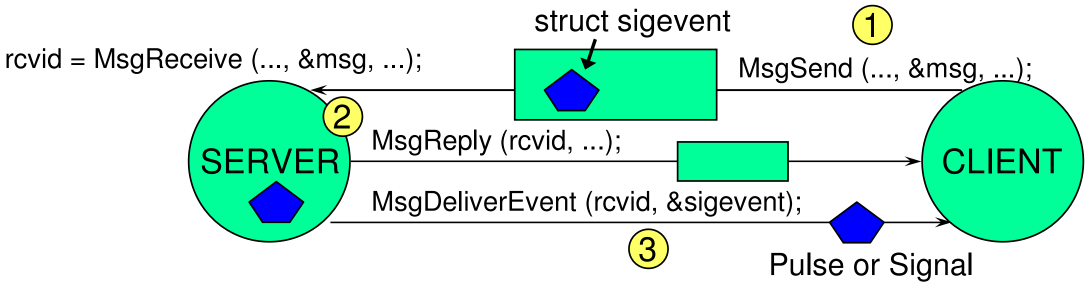


- timer events work in a similar fashion
- timer and interrupt handling are covered further in their respective course sections

NOTES:

Event Delivery Example

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

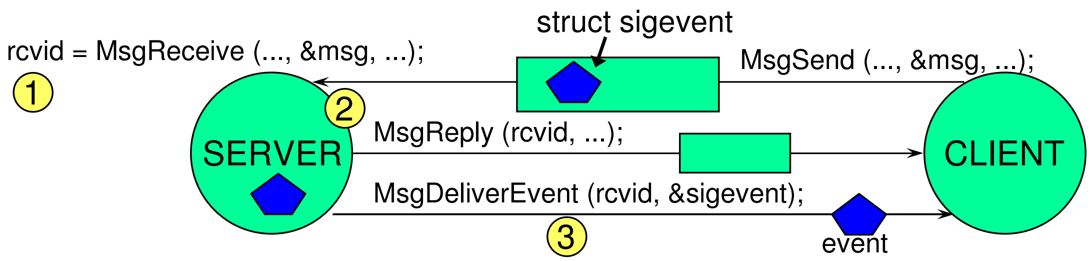
   ☞ server never needs to know what form the event will take, *MsgDeliverEvent()* takes care of it

NOTES:

In step 1, the client is actually sending a message that encapsulates the event description.

Event Delivery - the rcvid

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

NOTES:

Again, the message the client sends contains a **`sigevent`**, and usually identification as to when the event should be delivered. The server extracts and saves the **`sigevent`** along with the **`rcvid`**.

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

NOTES:

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.

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

continued...

NOTES:

Exercise (continued):

- To make it easier, searching for the word “class” 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

NOTES:

Topics:

NOTES:

QNET:

- allows QNX native message passing over a network
  - code to message with server on another computer is same as local case
  - allows existing message passing apps to become network distributed
- is implemented by a module which plugs into the QNX network stack, **`io-pkt-*`**
- can be ethernet or IP encapsulated
  - if IP encapsulated, it is routable
  - could be encapsulated on other protocols with custom **`io-pkt`** plug-ins
- can be enabled by:

```
mount -T io-pkt lsm-qnet.so
```


NOTES:

Overview

QNET:

- QNET causes namespaces on other nodes to be available in **/net**
  - this is a configurable name, **net** is the default
- this allows resources to be shared across the network
- from code, name resolutions are done (almost) the same way for local or network:

local
```
fd = open("/dev/Null",…);
```
net
```
fd = open("/net/ctrl-node/dev/Null",…);
```

local
```
coid = name_open("myname",0);
```
net
```
coid = name_open("myname",
        NAME_ATTACH_FLAG_GLOBAL);
```

- name location (*name_open()*) across the network requires the Global Name Service (**gns**) processes

NOTES:

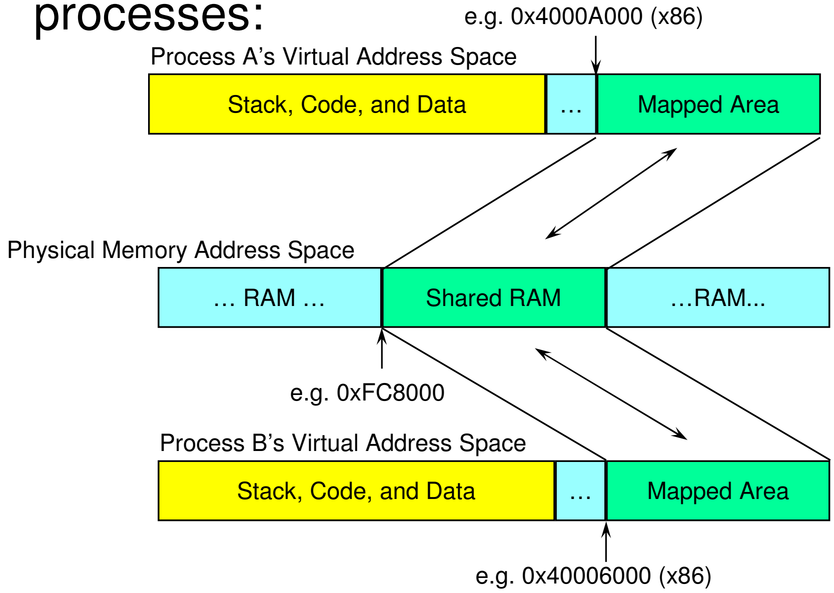
Topics:

- 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
- QNET
- → Shared Memory
- PPS
- Conclusion

NOTES:

Shared Memory

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



NOTES:

There is no expectation or guarantee that physical addresses in a shared area would be contiguous (as the diagram suggests) though they might be.

To set up shared memory:

`fd = shm_open( "/myname", O_RDWR|O_CREAT, 0666 );`

- 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

NOTES:

To access a shared memory object:

```
fd = shm_open( "/myname",O_RDWR, 0666);
```

– 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 pagesize boundaries, even if offset and size aren't

```
close(fd);
```

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

NOTES:

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
```


NOTES:

Problems with shared memory:

- readers don't know when data is stable
- writers don't know when it is safe to write

These are synchronization problems.

Let's look at a few solutions...

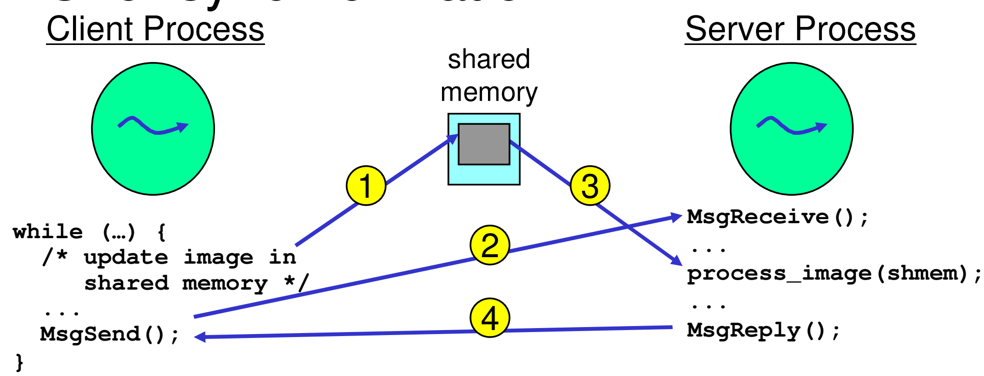
NOTES:

There are a variety of synchronization solutions:

- thread synchronization objects in the shared memory area
  - if using *sem_init()* the **pshared** parameter must be non-zero
  - mutexes and condition variables need the **PTHREAD_PROCESS_SHARED** flag
- *atomic_*()* functions for control variables
- IPC
  - *MsgSend()/MsgReceive()/MsgReply()* has built-in synchronization
  - use the shared memory to avoid large data copies

NOTES:

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.

NOTES:

Exercise:

- in the **`ipc`** project:
- **`shmemcreator.c`**, **`shmemuser.c`** and **`shmem.h`** demonstrate using shared memory
- they use a semaphore for passing control
- to run:

```
shmemcreator /myname
shmemuser /myname
```

- examine the code

NOTES:

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
- QNET
- Shared Memory
- → PPS
- Conclusion

NOTES:

Persistent Publish and Subscribe:

- is a low volume, one-to-many IPC mechanism
- a publisher supplies data
- subscriber(s) are given the data, or notified that the data is available
- the **`pps`** manager handles the data, notification, and persistence
- the API is (primarily) POSIX file access: *open()*, *read()*, *write()*, *select()*, etc
- is used by much of the QNX HMI framework

NOTES:

The **`pps`** server:

- manages the names in the pathname space
  - current implementation registers at **`/pps`** by default
- requires an underlying file system for the persistent storage
  - e.g. a disk or flash based file system
- stores data/attributes when a publisher writes them to an object
- notifies subscribers when data on an object has changed

NOTES:

By default, the current implementation writes the persistent data to the directory /var/pps, and that directory needs to be created in advance.

Objects and Attributes

- each file is an object
- each object will have attributes associated with it
  - attributes may be created, deleted, or modified
  - more than one publisher can change the same attribute, or different attributes of the same object
  - an attribute is a string:
    - <attribute_name>:<encoding>:<value>\n
      - name and encoding may contain alpha-numerics or underscore and period
      - value can be any characters except null or linefeed
- can easily be read with **`cat`** for debugging

NOTES:

PPS - subscriber

A subscriber:

- opens the object for READ
- may request notification of attribute changes:
  - *ionotify()* can request a signal or pulse on change
  - *select()* can be used to block until change on one or more objects or other fds
- a read() will return the current object state
- reads default to non-blocking, treating the object like a file
  - can be set to blocking with *fcntl()* or by opening the object **`"?wait"`** appended to the name

NOTES:

PPS - subscriber

A read() will return:

– object name, and attributes, general format:

```
@<object_name>
<attr1_name>:<encoding>:<attribute_one_value>
<attr2_name>:<encoding>:<attribute_two_value>
```

– e.g.

```
@Current_Position
lat::45.417266
long::-75.696895
address::221 Bank Street
city::Ottawa,Ontario,Canada
```

– the encoding, meaning, and parsing of the attributes is user-defined

NOTES:

PPS - publisher

Data is published with *write()*:

- one or multiple attributes can be changed with a single *write()*
- an attribute can be deleted by preceding its name with a minus sign, e.g.:
  - **`echo "-address" >> /pps/Current_Position`**
- all attributes of an object are deleted if the object is open **`O_TRUNC`**
  - **`echo anything > /pps/object`**

  will do this!
- all subscribers will be notified of changes as appropriate

NOTES:

PPS - example

Example:

- make sure **`/var/pps`** exists if you want persistent objects
- run the **`pps`** server
- in your **`ipc_pps`** project:
  - **`pps_publisher`** will create and publish to the object **`/pps/count`**
  - **`pps_client_select`** will block on *select()* and read data when notified
  - **`pps_client_notify`** will get pulse notifications when there are changes
- you can also look or modify the object with command line tools like **`cat`** and **`echo`**

NOTES:

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
- QNET
- Shared Memory
- PPS
- → Conclusion

NOTES:

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

NOTES: